VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Acceleration of the space-time boundary element method using GPUs

Akcelerace prostoro-časové metody hraničních prvků pomocí GPU

Jakub Homola

Diploma thesis

Supervisor: Ing. Michal Merta, Ph.D.

Ostrava, 2021

Abstrakt

V této diplomové práci se zabýváme akcelerací prostoro-časové metody hraničních prvků pro řešení rovnice tepla za použití grafických akcelerátorů. Tato metoda, narozdíl od sekvenčního procházení časových kroků, sestavuje globální prostoro-časové matice, které mají velké nároky na paměť. To omezuje velikost problémů, které jsme tímto přístupem schopni řešit. Vycházíme z existující CPU implementace knihovny BESTHEA, kterou rozšíříme o GPU-akcelerovaný kód pro násobení matice-vektor, který počítá prvky matice za běhu až když jsou potřeba. Protože matice nemusejí být uloženy v paměti, umožňuje tento přístup řešit velké problémy i na GPU akcelerátorech s limitovanou kapacitou paměti. Tímto přístupem jsme oproti původnímu CPU kódu dosáhli zrychlení v řádu desítek a byli jsme schopni řešit daleko větší problémy.

Klíčová slova: metoda hraničních prvků, BEM, prostoro-časová metoda hraničních prvků, rovnice tepla, BESTHEA, grafické akcelerátory, GPU, CUDA

Abstract

In this thesis we aim at accelerating the space-time boundary element method for the heat equation using GPUs. Contrary to the time-stepping approaches, the method assembles the global space-time system matrices, which have large memory requirements. This limits the size of problems that can be solved. Starting from the existing CPU implementation in the BESTHEA library, we develop a GPU-accelerated code that computes the matrix values during the matrix-vector multiplication on the fly as they are needed. This enables us to solve large problems even on GPU accelerators with limited amount of memory, since the matrices do not have to be assembled and stored. Using this approach we achieved a speedup in the order of tens with respect to the original CPU code and were able to solve significantly larger problems.

Key words: boundary element method, BEM, space-time boundary element method, heat equation, BESTHEA, GPU, CUDA

I would like to thank my supervisor, Ing. Michal Merta, Ph.D., for the opportunity to work on such an interesting topic and for all the help with writing this thesis.

Contents

List of symbols and abbreviations

BEM	–	boundary element method
FEM	–	finite element method
BESTHEA	–	`C++` library, Boundary Element Method for The Heat EquAtion
OpenMP	–	Open Multi-Processing
Intel MKL	–	Intel Math Kernel Library
FGMRES	–	Flexible Generalized Minimal RESidual (method)
GPU	–	Graphics Processing Unit
CPU	–	Central Processing Unit
α	–	heat capacity constant
$\mathbb{R}$	–	Set of all real numbers
Ω	–	bounded Lipschitz domain
T	–	end time
Q	–	space-time domain, space-time cylinder
G_α	–	fundamental solution of the heat equation in 3 spatial dimensions
Σ	–	lateral surface of the space-time cylinder
γ_0	–	Dirichlet trace operator
γ_1	–	Neumann trace operator
X, X^*	–	anisotropic Sobolev spaces
V	–	single layer operator
K	–	double layer operator
K'	–	adjoint double layer operator
D	–	hypersingular operator
I	–	identity operator
E_t	–	number of temporal elements
E_s	–	number of spatial elements
N_s	–	number of spatial nodes
$\mathcal{T}_h$	–	discretized time interval
Γ_h	–	discretized boundary of Ω
Σ_h	–	discretized lateral surface of the space-time cylinder
h_t	–	timestep length
t_i	–	i-th timestep
γ_j	–	j-th spatial element
μ_j	–	j-th spatial node
$X_h^{0,0}$	–	discretized Sobolev space X^*
$X_h^{0,1}$	–	discretized Sobolev space X
V_h	–	single layer matrix
K_h	–	double layer matrix
$\mathsf{K}_h^{\top_s}$	–	adjoint double layer matrix
D_h	–	hypersingular matrix
M_h	–	mass matrix
erf	–	error function
Δ_j	–	surface area of the j-th element
$\widetilde{V}$	–	discretized single layer operator
W	–	discretized double layer operator

List of Figures

List of Tables

List of Listings

1 Introduction

Boundary element methods (BEM) [21,23] for solving partial differential equations have several advantages over the finite element methods (FEM). We need to discretize and solve the problem only on the boundary of the domain, which significantly reduces the size of the problem. BEM is also well-suited for problems stated on unbounded domains. The matrices arising from BEM have smaller dimensions, but are fully populated, thus having large memory requirements.

In this thesis we deal with the space-time boundary element method for the heat equation, extending the boundary element methods with a temporal dimension. Conventional approaches to solving the heat equation calculate the solution sequentially in small timesteps, exploiting parallelism only in the spatial domain. The space-time approach deals with the discretized time interval as a whole, enabling parallelization in both space and time. This, however, increases the memory requirements even more.

In this thesis we work with and extend the functionality of the BESTHEA library[1] (*Boundary Element Solver for The Heat EquAtion* [17]) developed in `C++`. It contains classes and functions enabling its user to solve the space-time boundary element method for the heat equation efficiently and in parallel. The current implementation assembles the matrices and stores them in memory and therefore has large memory requirements. This limits the size of the problems we are able to solve using the library.

To overcome this limitation we do not to store the matrices in memory, but rather calculate the matrix entries during matrix-vector multiplication on the fly as they are needed. The performance penalty of calculating the matrix entries during every matrix-vector multiplication is very large, especially when the matrix is used in an iterative solver. However, using the massive computational power of today's GPUs should partly negate this issue.

The core objective of this thesis is therefore to implement an algorithm that performs on-the-fly matrix-vector multiplication using GPUs, where the matrices arise from the space-time boundary element method for the heat equation. We first create a CPU version of the algorithm, which we then accelerate with CUDA. The developed code is a part of the BESTHEA library, which will be accessible as open source (`https://github.com/zap150/besthea`).

This text is structured as follows. In the second section we describe a boundary integral formulation of the heat equation and its discretization using the space-time boundary element method. In Section 3 we introduce the BESTHEA library and go through its current internal functionality. In the fourth section we mention several approaches to utilizing GPUs and make a brief overview of CUDA. In Section 5 we explain the techniques and algorithms we used to accelerate the library. In the final section we conduct several numerical experiments focused mainly on performance of different implementation approaches.

[1]Development of the BESTHEA library was supported by the Czech Science Foundation under the project 17-22615S and by the Austrian Science Foundation under the project I4033-N32

2 Space-time boundary element method for the heat equation

In this section we describe a boundary integral formulation of the heat equation and its discretization using the space-time boundary element method. In what follows we draw mainly from [14, 30].

Let $\Omega \subset \mathbb{R}^3$ be a bounded Lipschitz domain, $T \in \mathbb{R}^+$ the end time, $Q \coloneqq \Omega \times (0,T)$ the space-time cylinder and $\Sigma \coloneqq \partial\Omega \times (0,T)$ the lateral surface of the space-time cylinder (see Figure 1). We aim to solve the heat equation

$$\frac{\partial u}{\partial t}(\boldsymbol{x},t) - \alpha \Delta u(\boldsymbol{x},t) = 0 \quad \text{for } (\boldsymbol{x},t) \in Q,$$

where $\alpha > 0$ is a heat capacity constant, together with an initial condition (which we for simplicity assume to be zero)

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \equiv 0 \quad \text{for } \boldsymbol{x} \in \Omega$$

and either Dirichlet

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \text{for } (\boldsymbol{x},t) \in \Sigma,$$

or Neumann boundary condition

$$\alpha \frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{x},t) = h(\boldsymbol{x},t) \quad \text{for } (\boldsymbol{x},t) \in \Sigma.$$

2.1 Boundary integral equations, variational formulations

Using the representation formula, the solution u can be for all $(\boldsymbol{x},t) \in Q$ expressed only using the values of u and $\frac{\partial u}{\partial \boldsymbol{n}}$ on Σ

$$\begin{aligned} u(\boldsymbol{x},t) = &\overbrace{\int_\Omega G_\alpha(\boldsymbol{x}-\boldsymbol{y},t) u_0(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{y}}^{=0} \\ &+ \int_0^t \int_{\partial\Omega} G_\alpha(\boldsymbol{x}-\boldsymbol{y},t-\tau) \alpha \frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{y},\tau) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\tau \\ &- \int_0^t \int_{\partial\Omega} \alpha \frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau) u(\boldsymbol{y},\tau) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\tau. \end{aligned} \tag{1}$$

The terms on the right-hand side are called the initial, single layer, and double layer potential, respectively (with the initial potential being zero because of the zero initial condition), and G_α is the fundamental solution to the heat equation in 3 spatial dimensions,

$$G_\alpha(\boldsymbol{x}-\boldsymbol{y},t-\tau) = \begin{cases} \dfrac{1}{(4\pi\alpha(t-\tau))^{3/2}} \exp\left(-\dfrac{\|\boldsymbol{x}-\boldsymbol{y}\|^2}{4\alpha(t-\tau)}\right) & \text{for } t > \tau, \\ 0 & \text{otherwise,} \end{cases}$$

having the scaled normal derivative

$$\alpha \frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau) = \begin{cases} \dfrac{(\boldsymbol{x}-\boldsymbol{y})\cdot \boldsymbol{n}_{\boldsymbol{y}}}{16(\pi\alpha)^{3/2}(t-\tau)^{5/2}} \exp\left(-\dfrac{\|\boldsymbol{x}-\boldsymbol{y}\|^2}{4\alpha(t-\tau)}\right) & \text{for } t > \tau, \\ 0 & \text{otherwise.} \end{cases}$$

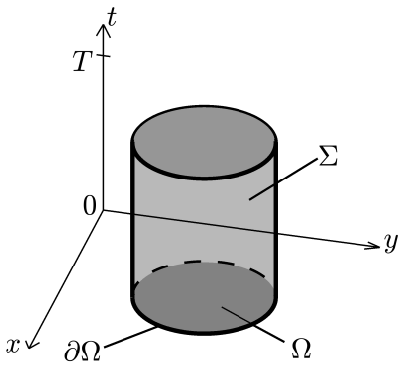


Figure 1: Visualization of the space-time cylinder in two spatial dimensions and time

Applying the Dirichlet trace operator [4, 10]

$$\gamma_0(v)(\boldsymbol{y},t) = \lim_{\tilde{\boldsymbol{y}}\in\Omega,\tilde{\boldsymbol{y}}\to\boldsymbol{y}\in\partial\Omega} v(\tilde{\boldsymbol{y}},t) \quad \text{for } (\boldsymbol{y},t)\in\Sigma$$

to the representation formula (1) we obtain the first boundary integral equation,

$$\begin{aligned}\frac{1}{2}u(\boldsymbol{x},t) = &\int_0^t\int_{\partial\Omega} G_\alpha(\boldsymbol{x}-\boldsymbol{y},t-\tau)\alpha\frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{y},\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau\\ &-\int_0^t\int_{\partial\Omega}\alpha\frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau)u(\boldsymbol{y},\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau \quad \text{for } (\boldsymbol{x},t)\in\Sigma.\end{aligned} \tag{2}$$

By applying the Neumann trace operator

$$\gamma_1(v)(\boldsymbol{y},t) = \lim_{\tilde{\boldsymbol{y}}\in\Omega,\tilde{\boldsymbol{y}}\to\boldsymbol{y}\in\partial\Omega} \alpha n(\boldsymbol{y})\cdot\nabla_{\tilde{\boldsymbol{y}}} v(\tilde{\boldsymbol{y}},t) \quad \text{for } (\boldsymbol{y},t)\in\Sigma$$

we obtain the second boundary integral equation

$$\begin{aligned}\frac{1}{2}\alpha\frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{x},t) = &\int_0^t\int_{\partial\Omega}\alpha\frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{x}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau)\alpha\frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{y},\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau\\ &+\alpha\frac{\partial}{\partial \boldsymbol{n}_{\boldsymbol{x}}}\int_0^t\int_{\partial\Omega}\alpha\frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau)u(\boldsymbol{y},t)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau \quad \text{for } (\boldsymbol{x},t)\in\Sigma.\end{aligned} \tag{3}$$

Let us denote $X = H^{1/2,1/4}(\Sigma)$ and its dual $X^* = H^{-1/2,-1/4}(\Sigma)$. For definition of these anisotropic Sobolev spaces see [11]. For all $(\boldsymbol{x},t)\in\Sigma$ we define the following operators

$$\begin{aligned}V : X^*\to X,\qquad & V(w)(\boldsymbol{x},t) = \int_0^t\int_{\partial\Omega} G_\alpha(\boldsymbol{x}-\boldsymbol{y},t-\tau)w(\boldsymbol{y},\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau,\\ K : X\to X,\qquad & K(u)(\boldsymbol{x},t) = \int_0^t\int_{\partial\Omega}\alpha\frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau)u(\boldsymbol{y},\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau,\\ K' : X^*\to X^*,\qquad & K'(w)(\boldsymbol{x},t) = \int_0^t\int_{\partial\Omega}\alpha\frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{x}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau)w(\boldsymbol{y},\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau,\\ D : X\to X^*,\qquad & D(u)(\boldsymbol{x},t) = \alpha\frac{\partial}{\partial \boldsymbol{n}_{\boldsymbol{x}}}\int_0^t\int_{\partial\Omega}\alpha\frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y},t-\tau)u(\boldsymbol{y},t)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau.\end{aligned}$$

These are, in the order given, the single layer, double layer, adjoint double layer, and hypersingular operators. Rearranging the terms in (2) and (3) and using the above-defined operators we obtain for $(\boldsymbol{x}, t) \in \Sigma$

$$V(w)(\boldsymbol{x}, t) = \left(\frac{1}{2}I + K\right)(u)(\boldsymbol{x}, t),$$
$$D(u)(\boldsymbol{x}, t) = \left(\frac{1}{2}I - K'\right)(w)(\boldsymbol{x}, t),$$

respectively, where $w \coloneqq \alpha\frac{\partial u}{\partial \boldsymbol{n}}$. Replacing u and w on the right-hand side with the known boundary condition functions g and h we get for $(\boldsymbol{x}, t) \in \Sigma$

$$V(w)(\boldsymbol{x}, t) = \left(\frac{1}{2}I + K\right)(g)(\boldsymbol{x}, t), \tag{4}$$
$$D(u)(\boldsymbol{x}, t) = \left(\frac{1}{2}I - K'\right)(h)(\boldsymbol{x}, t). \tag{5}$$

The boundary integral equations (4) and (5) are equivalent to the variational formulations

$$\forall q \in X^* : \quad \langle V(w), q\rangle_\Sigma = \left\langle \left(\frac{1}{2}I + K\right)(g), q \right\rangle_\Sigma, \tag{6}$$
$$\forall r \in X : \quad \langle D(u), r\rangle_\Sigma = \left\langle \left(\frac{1}{2}I - K'\right)(h), r \right\rangle_\Sigma, \tag{7}$$

where

$$\langle v, w\rangle_\Sigma \coloneqq \int_0^T \int_{\partial\Omega} v(\boldsymbol{x}, t) w(\boldsymbol{x}, t)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t.$$

In the case of the hypersingular operator we use the equivalent representation [18]

$$\begin{aligned}\langle D(u), r\rangle_\Sigma = \alpha^2 \int_0^T \int_{\partial\Omega} \mathbf{curl}_{\partial\Omega} r(\boldsymbol{x}, t)^\top \int_0^t \int_{\partial\Omega} \mathbf{curl}_{\partial\Omega} u(\boldsymbol{y}, \tau) G_\alpha(\boldsymbol{x} - \boldsymbol{y}, t - \tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t \\ - \alpha \int_0^T \int_{\partial\Omega} \boldsymbol{n}(\boldsymbol{x})^\top r(\boldsymbol{x}, t) \int_0^t \int_{\partial\Omega} \boldsymbol{n}(\boldsymbol{y}) u(\boldsymbol{y}, t) \frac{\partial G_\alpha}{\partial \tau}(\boldsymbol{x} - \boldsymbol{y}, t - \tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t.\end{aligned}$$

2.2 Discretization

We divide the time interval $(0, T)$ into E_t uniformly spaced elements $\mathcal{T}_h = \{(t_{i-1}, t_i)\}_{i=1}^{E_t}$, where $t_i = ih_t$ and $h_t = T/E_t$, so that

$$\overline{(0, T)} = \bigcup_{i=1}^{E_t} \overline{(t_{i-1}, t_i)}.$$

The spatial surface $\partial\Omega$ is discretized into a triangular surface mesh Γ_h consisting of E_s elements $\{\gamma_j\}_{j=1}^{E_s}$ and N_s nodes $\{\mu_j\}_{j=1}^{N_s}$,

$$\Gamma_h = \bigcup_{j=1}^{E_s} \overline{\gamma_j}.$$

We define the discretized lateral surface of the space-time cylinder $\Sigma_h \coloneqq \Gamma_h \times \mathcal{T}_h$.

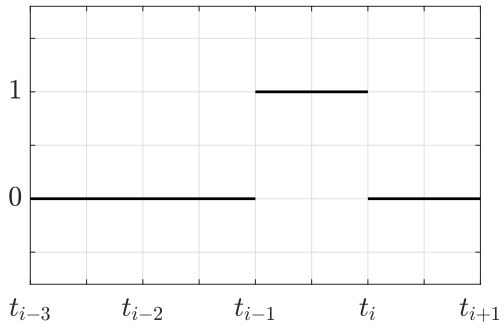


Figure 2: Illustration of function $\varphi^0_{t,i}$

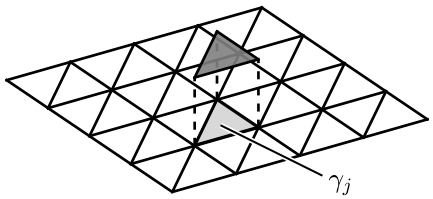


Figure 3: Illustration of function $\varphi^0_{s,j}$

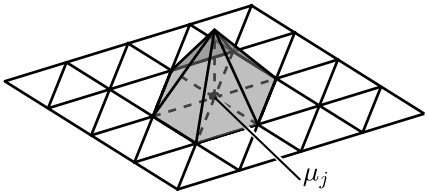


Figure 4: Illustration of function $\varphi^1_{s,j}$

For all $i \in \{1, 2, \ldots, E_t\}$ we define a function $\varphi^0_{t,i}$ piecewise constant in time (see Figure 2)

$$\varphi^0_{t,i}(t) = \begin{cases} 1 & t \in (t_{i-1}, t_i), \\ 0 & \text{otherwise}, \end{cases}$$

for all $j \in \{1, 2, \ldots, E_s\}$ we define a function $\varphi^0_{s,j}$ piecewise constant on the discretized boundary Γ_h (illustrated in Figure 3)

$$\varphi^0_{s,j}(\boldsymbol{x}) = \begin{cases} 1 & \boldsymbol{x} \in \gamma_j, \\ 0 & \text{otherwise}, \end{cases}$$

and for all $j \in \{1, 2, \ldots, N_s\}$ we define a function $\varphi^1_{s,j}$ globally continuous and piecewise linear on the discretized boundary Γ_h such that for all $m \in \{1, 2, \ldots, N_s\}$

$$\varphi^1_{s,j}(\mu_m) = \delta_{j,m} = \begin{cases} 1 & m = j, \\ 0 & m \neq j. \end{cases}$$

An example of such function is illustrated in Figure 4.

We define the discretized spaces $X_h^{0,0}$ and $X_h^{0,1}$ approximating the spaces X^* and X, respectively, as

$$X_h^{0,0} := \operatorname{span}(\{\varphi^{0,0}_{ts,k}\}_{k=1}^{E_t E_s}),$$
$$X_h^{0,1} := \operatorname{span}(\{\varphi^{0,1}_{ts,k}\}_{k=1}^{E_t N_s}),$$

where the basis functions $\varphi^{0,0}_{ts,k}$ and $\varphi^{0,1}_{ts,k}$ are defined as

$$\varphi^{0,0}_{ts,k}(\boldsymbol{x},t) \coloneqq \varphi^{0}_{t,i}(t)\varphi^{0}_{s,j}(\boldsymbol{x}),$$
$$\varphi^{0,1}_{ts,k}(\boldsymbol{x},t) \coloneqq \varphi^{0}_{t,i}(t)\varphi^{1}_{s,j}(\boldsymbol{x}),$$

with the index mapping $k = iE_s + j$ for $\varphi^{0,0}_{ts,k}$ and $k = iN_s + j$ for $\varphi^{0,1}_{ts,k}$. The space $X^{0,0}_h$ therefore contains functions piecewise constant both in space and time, while the space $X^{0,1}_h$ contains functions globally continuous piecewise linear in space and piecewise constant in time.

We approximate $w \in X$ and $u \in X^*$ with functions $w_h \in X^{0,0}_h$ and $u_h \in X^{0,1}_h$, which can be written as a linear combination of the basis functions

$$w_h(\boldsymbol{x},t) = \sum_{k=1}^{E_tE_s} w_k\varphi^{0,0}_{ts,k}(\boldsymbol{x},t) = \sum_{i=1}^{E_t}\sum_{j=1}^{E_s} w_{i,j}\varphi^{0}_{t,i}(t)\varphi^{0}_{s,j}(\boldsymbol{x}), \tag{8}$$

$$u_h(\boldsymbol{x},t) = \sum_{k=1}^{E_tN_s} u_k\varphi^{0,1}_{ts,k}(\boldsymbol{x},t) = \sum_{i=1}^{E_t}\sum_{j=1}^{N_s} u_{i,j}\varphi^{0}_{t,i}(t)\varphi^{1}_{s,j}(\boldsymbol{x}), \tag{9}$$

where $\boldsymbol{w} \in \mathbb{R}^{E_tE_s}$ and $\boldsymbol{u} \in \mathbb{R}^{E_tN_s}$.

The boundary condition functions $g \in X^*$ and $h \in X$ are orthogonally projected onto the discretized spaces $X^{0,1}_h$ and $X^{0,0}_h$, yielding the functions g_h and h_h with basis coordinate vectors $\boldsymbol{g} \in \mathbb{R}^{E_tN_s}$ and $\boldsymbol{h} \in \mathbb{R}^{E_tE_s}$, respectively. E.g. in the case of Dirichlet boundary condition we search for $g_h \in X^{0,1}_h$ such that

$$g_h = \operatorname*{arg\,min}_{\tilde{g}_h \in X^{0,1}_h} \frac{1}{2}\|\tilde{g}_h - g\|_{L^2(\Sigma)},$$

which is equivalent to solving the system

$$\sum_{k=1}^{E_tN_s} g_k h_t \langle \varphi^{0,1}_{ts,k}, \varphi^{0,1}_{ts,l}\rangle_\Sigma = \langle g, \varphi^{0,1}_{ts,l}\rangle_\Sigma \qquad \forall \ell \in \{1,2,\dots,E_tN_s\}.$$

2.3 Systems of linear equations, boundary element matrices

Plugging the approximations (8) and (9) into the variational formulations (6) and (7), using the discretized boundary condition functions $g_h \approx g$ and $h_h \approx h$ and testing with all basis functions $\varphi^{0,0}_{ts,l}$ and $\varphi^{0,1}_{ts,l}$, respectively, we get

$$\begin{aligned}\forall l \in \{1,2,\dots,E_tE_s\}: \quad \sum_{k=1}^{E_tE_s} w_k\langle V(\varphi^{0,0}_{ts,k}), \varphi^{0,0}_{ts,l}\rangle_{\Sigma_h} &= \frac{1}{2}\sum_{k=1}^{E_tN_s} g_k\langle \varphi^{0,1}_{ts,k}, \varphi^{0,0}_{ts,l}\rangle_{\Sigma_h} \\ &\quad + \sum_{k=1}^{E_tN_s} g_k\langle K(\varphi^{0,1}_{ts,k}), \varphi^{0,0}_{ts,l}\rangle_{\Sigma_h}\end{aligned} \tag{10}$$

for the first boundary integral equation and

$$\begin{aligned}\forall l \in \{1,2,\dots,E_tN_s\}: \quad \sum_{k=1}^{E_tN_s} u_k\langle D(\varphi^{0,1}_{ts,k}), \varphi^{0,1}_{ts,l}\rangle_{\Sigma_h} &= \frac{1}{2}\sum_{k=1}^{E_tE_s} h_k\langle \varphi^{0,0}_{ts,k}, \varphi^{0,1}_{ts,l}\rangle_{\Sigma_h}, \\ &\quad - \sum_{k=1}^{E_tE_s} h_k\langle K'(\varphi^{0,0}_{ts,k}), \varphi^{0,1}_{ts,l}\rangle_{\Sigma_h}\end{aligned} \tag{11}$$

for the second boundary integral equation. This leads to the systems of linear equations

$$\mathsf{V}_h \boldsymbol{w} = \frac{1}{2}\mathsf{M}_h \boldsymbol{g} + \mathsf{K}_h \boldsymbol{g}, \tag{12}$$

$$\mathsf{D}_h \boldsymbol{u} = \frac{1}{2}\mathsf{M}_h^\top \boldsymbol{h} - \mathsf{K}_h^{\top_s} \boldsymbol{h}, \tag{13}$$

with the following block matrices with block dimensions[2] $E_t \times E_t$:

$$\begin{aligned}
\mathsf{V}_h[l,k] &= \langle V(\varphi^{0,0}_{ts,k}), \varphi^{0,0}_{ts,l}\rangle_{\Sigma_h} &&= \langle V(\varphi^0_{t,i_c}\varphi^0_{s,j_c}), \varphi^0_{t,i_r}\varphi^0_{s,j_r}\rangle_{\Sigma_h} &&= \mathsf{V}_h[i_r,i_c][j_r,j_c],\\
\mathsf{K}_h[l,k] &= \langle K(\varphi^{0,1}_{ts,k}), \varphi^{0,0}_{ts,l}\rangle_{\Sigma_h} &&= \langle K(\varphi^0_{t,i_c}\varphi^1_{s,j_c}), \varphi^0_{t,i_r}\varphi^0_{s,j_r}\rangle_{\Sigma_h} &&= \mathsf{K}_h[i_r,i_c][j_r,j_c],\\
\mathsf{K}_h^{\top_s}[l,k] &= \langle K'(\varphi^{0,0}_{ts,k}), \varphi^{0,1}_{ts,l}\rangle_{\Sigma_h} &&= \langle K'(\varphi^0_{t,i_c}\varphi^0_{s,j_c}), \varphi^0_{t,i_r}\varphi^1_{s,j_r}\rangle_{\Sigma_h} &&= \mathsf{K}_h^{\top_s}[i_r,i_c][j_r,j_c],\\
\mathsf{D}_h[l,k] &= \langle D(\varphi^{0,1}_{ts,k}), \varphi^{0,1}_{ts,l}\rangle_{\Sigma_h} &&= \langle D(\varphi^0_{t,i_c}\varphi^1_{s,j_c}), \varphi^0_{t,i_r}\varphi^1_{s,j_r}\rangle_{\Sigma_h} &&= \mathsf{D}_h[i_r,i_c][j_r,j_c],\\
\mathsf{M}_h[l,k] &= \langle \varphi^{0,1}_{ts,k}, \varphi^{0,0}_{ts,l}\rangle_{\Sigma_h} &&= \langle \varphi^0_{t,i_c}\varphi^1_{s,j_c}, \varphi^0_{t,i_r}\varphi^0_{s,j_r}\rangle_{\Sigma_h} &&= \mathsf{M}_h[i_r,i_c][j_r,j_c],\\
\mathsf{M}_h^\top[l,k] &= \langle \varphi^{0,0}_{ts,k}, \varphi^{0,1}_{ts,l}\rangle_{\Sigma_h} = \langle \varphi^{0,1}_{ts,l}, \varphi^{0,0}_{ts,k}\rangle_{\Sigma_h} &&= \mathsf{M}_h[k,l],
\end{aligned}$$

where we again used an appropriate index mapping. We index the matrices with two pairs of indices, first of which specifies position of a block in the matrix, while the second pair specifies location of an entry within the block. The matrices are collectively called the boundary element matrices. V_h, K_h, $\mathsf{K}_h^{\top_s}$ and D_h are the single layer, double layer, adjoint double layer and hypersingular matrices, respectively, and we will collectively call them the main boundary element matrices. M_h and $\mathsf{M}_h^\top$ are usually called the mass matrices and they represent the discretized identity operators.

2.3.1 Single layer matrix V_h

We start with breaking down the formula for an entry of the matrix V_h. Observing that $\operatorname{supp}\varphi^0_{t,i} = (t_{i-1}, t_i)$ and $\operatorname{supp}\varphi^0_{s,j} = \gamma_j$, we can write [14, 30]

$$\begin{aligned}
\mathsf{V}_h[i_r,i_c][j_r,j_c] &= \langle V(\varphi^0_{t,i_c}\varphi^0_{s,j_c}), \varphi^0_{t,i_r}\varphi^0_{s,j_r}\rangle_{\Sigma_h}\\
&= \int_0^T \int_{\Gamma_h} \left(\int_0^t \int_{\Gamma_h} G_\alpha(\boldsymbol{x}-\boldsymbol{y}, t-\tau)\varphi^0_{t,i_c}(\tau)\varphi^0_{s,j_c}(\boldsymbol{y})\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau \right) \varphi^0_{t,i_r}(t)\varphi^0_{s,j_r}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t\\
&= \int_{t_{i_r-1}}^{t_{i_r}} \int_{\gamma_{j_r}} \left(\int_0^t \int_{\gamma_{j_c}} G_\alpha(\boldsymbol{x}-\boldsymbol{y}, t-\tau)\varphi^0_{t,i_c}(\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau \right) \mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t\\
&= \begin{cases}
\displaystyle\int_{t_{i_r-1}}^{t_{i_r}} \int_{\gamma_{j_r}} \int_{t_{i_c-1}}^{t_{i_c}} \int_{\gamma_{j_c}} G_\alpha(\boldsymbol{x}-\boldsymbol{y}, t-\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t & \text{for } i_c < i_r,\\
\displaystyle\int_{t_{i_r-1}}^{t_{i_r}} \int_{\gamma_{j_r}} \int_{t_{i_c-1}}^{t} \int_{\gamma_{j_c}} G_\alpha(\boldsymbol{x}-\boldsymbol{y}, t-\tau)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\tau\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}\,\mathrm{d}t & \text{for } i_c = i_r,\\
0 & \text{for } i_c > i_r.
\end{cases}
\end{aligned}$$

Notice, that the fundamental solution G_α only depends on the difference $t-\tau$, therefore we can shift both t and τ by the same value without changing the result of the integral. We subtract

[2] by block dimensions we mean the number of block rows and columns of the matrix

t_{i_c-1} in both the first and second case and denote $d = i_r - i_c$, obtaining

$$\mathsf{V}_h^d[j_r, j_c] = \mathsf{V}_h[i_r, i_c][j_r, j_c] = \begin{cases} \displaystyle\int_{\gamma_{jr}} \int_{\gamma_{jc}} \int_{t_d}^{t_{d+1}} \int_0^{h_t} G_\alpha(\boldsymbol{x} - \boldsymbol{y}, t - \tau) \,\mathrm{d}\tau \,\mathrm{d}t \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} & \text{for } d > 0 \\ \displaystyle\int_{\gamma_{jr}} \int_{\gamma_{jc}} \int_0^{h_t} \int_0^{t} G_\alpha(\boldsymbol{x} - \boldsymbol{y}, t - \tau) \,\mathrm{d}\tau \,\mathrm{d}t \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} & \text{for } d = 0 \\ 0 & \text{for } d < 0. \end{cases}$$

Therefore, we found the value of an entry in the matrix only depends on the difference $d = i_r - i_c$ and not on the specific values of i_r and i_c. Considering the indexing we used, this reveals that all the blocks on the same block diagonal are equal, i.e., the matrix has block-Toeplitz structure. We also found, that all the blocks above the main block diagonal are zero, leading to block lower triangular structure of the matrix. Furthermore, since the fundamental solution G_α only depends on the norm of the difference $\boldsymbol{x} - \boldsymbol{y}$, the blocks themselves are symmetric.

The matrix V_h therefore has block lower triangular Toeplitz structure with block dimensions $E_t \times E_t$,

$$\mathsf{V}_h = \begin{bmatrix} \mathsf{V}_h^0 & \mathsf{O} & \dots & \mathsf{O} \\ \mathsf{V}_h^1 & \mathsf{V}_h^0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathsf{O} \\ \mathsf{V}_h^{E_t-1} & \dots & \mathsf{V}_h^1 & \mathsf{V}_h^0 \end{bmatrix},$$

where each block V_h^d has dimensions $E_s \times E_s$. The entries of the matrix are calculated as

$$\mathsf{V}_h^d[j_r, j_c] = \int_{\gamma_{jr}} \int_{\gamma_{jc}} V^d(\boldsymbol{x} - \boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{14}$$

where

$$V^d(\boldsymbol{r}) = \begin{cases} \displaystyle\int_0^{h_t} \int_0^{t} G_\alpha(\boldsymbol{r}, t - \tau) \,\mathrm{d}\tau \,\mathrm{d}t & \text{for } d = 0, \\ \displaystyle\int_{t_d}^{t_{d+1}} \int_0^{h_t} G_\alpha(\boldsymbol{r}, t - \tau) \,\mathrm{d}\tau \,\mathrm{d}t & \text{for } d \in \{1, 2, ..., E_t - 1\}. \end{cases}$$

The temporal integrals can be integrated analytically (for more details see [30]), leading to

$$\begin{aligned} V^0(\boldsymbol{r}) &= h_t G_\alpha^{\mathrm{d}\tau}(\boldsymbol{r}, 0) \quad + \; G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r}, 0) \quad - G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r}, h_t), \\ V^d(\boldsymbol{r}) &= -G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r}, (d-1)h_t) + 2G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r}, dh_t) - G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r}, (d+1)h_t), \end{aligned}$$

where

$$G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r}, \delta) = \frac{1}{4\pi} \left(\left(\frac{\|\boldsymbol{r}\|}{2\alpha^2} + \frac{\delta}{\alpha\|\boldsymbol{r}\|} \right) \operatorname{erf}\left(\frac{\|\boldsymbol{r}\|}{2\sqrt{\alpha\delta}} \right) + \frac{\sqrt{\delta}}{\sqrt{\pi\alpha^3}} \exp\left(-\frac{\|\boldsymbol{r}\|^2}{4\alpha\delta} \right) \right)$$

and

$$G_\alpha^{\mathrm{d}\tau}(\boldsymbol{r}, \delta) = \frac{1}{4\pi\alpha\|\boldsymbol{r}\|} \operatorname{erf}\left(\frac{\|\boldsymbol{r}\|}{2\sqrt{\alpha\delta}} \right),$$

with

$$\operatorname{erf}(x) \coloneqq \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \,\mathrm{d}t$$

being the error function. Due to singularities, we have to treat the following limiting cases:

$$\lim_{\delta\to 0+} G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r},\delta) = \frac{\|\boldsymbol{r}\|}{8\pi\alpha^2} \quad \text{for } \|\boldsymbol{r}\| > 0,$$

$$\lim_{\|\boldsymbol{r}\|\to 0+} G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{r},\delta) = \frac{\sqrt{\delta}}{2\sqrt{\pi^3\alpha^3}} \quad \text{for } \delta > 0,$$

$$\lim_{\delta\to 0+} G_\alpha^{\mathrm{d}\tau}(\boldsymbol{r},\delta) = \frac{1}{4\pi\alpha\|\boldsymbol{r}\|} \quad \text{for } \|\boldsymbol{r}\| > 0.$$

Knowing the results of temporal integration we can now rewrite (14) as

$$\begin{aligned}
\mathsf{V}_h^0[j_r,j_c] &= V_{S1}(j_r,j_c) &&+ V_{S2}(j_r,j_c) &&- V_R(j_r,j_c,1),\\
\mathsf{V}_h^1[j_r,j_c] &= -V_{S2}(j_r,j_c) &&+ 2V_R(j_r,j_c,1) &&- V_R(j_r,j_c,2),\\
\mathsf{V}_h^d[j_r,j_c] &= -V_R(j_r,j_c,d-1) &&+ 2V_R(j_r,j_c,d) &&- V_R(j_r,j_c,d+1) \quad \text{for } d\geq 2,
\end{aligned}$$

where

$$V_{S1}(j_r,j_c) = h_t \int_{\gamma_{j_r}}\int_{\gamma_{j_c}} G_\alpha^{\mathrm{d}\tau}(\boldsymbol{x}-\boldsymbol{y},0)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} = h_t \int_{\gamma_{j_r}}\int_{\gamma_{j_c}} \frac{1}{4\pi\alpha\|\boldsymbol{x}-\boldsymbol{y}\|}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{15}$$

$$V_{S2}(j_r,j_c) = \int_{\gamma_{j_r}}\int_{\gamma_{j_c}} G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{x}-\boldsymbol{y},0)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} = \int_{\gamma_{j_r}}\int_{\gamma_{j_c}} \frac{\|\boldsymbol{x}-\boldsymbol{y}\|}{8\pi\alpha^2}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{16}$$

$$V_R(j_r,j_c,d) = \int_{\gamma_{j_r}}\int_{\gamma_{j_c}} G_\alpha^{\mathrm{d}\tau\mathrm{d}t}(\boldsymbol{x}-\boldsymbol{y},dh_t)\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} \quad \text{for } d\geq 1. \tag{17}$$

We will call V_{S1} the first time-singular contribution, V_{S2} the second time-singular contribution and V_R the time-regular contribution. We will further split the naming of the time-regular contribution V_R in two, creating time-regular space-singular contribution (for identical elements, $j_r = j_c$) and fully regular contribution. This will be useful, because for identical test and trial elements we need to take care of the limiting case $\|\boldsymbol{r}\| \to 0$, while for nonidentical elements this is not necessary.

Notice, that for a given pair of row and column indices $[j_r, j_c]$ the values of the time-regular contribution V_R repeat themselves in neighboring blocks (consecutive values of d). We therefore do not need to evaluate them again in each block, instead the once calculated value of V_R can be used for multiple entries in the matrix. Similarly this holds for the values of V_{S2} and the first two blocks.

Similarly to the finite element method (FEM), we shift our view of the matrix assembly from "What is the value of this entry in the matrix?" to "How does this value of d and this pair of spatial elements contribute to the matrix?". We find, that for any ordered pair of spatial elements j_r and j_c (we call them test and trial elements, respectively) and for any $d \in \{1, 2, \ldots, E_t\}$ the value $V_R(j_r, j_c, d)$ contributes to entry $[j_r, j_c]$ in blocks $d-1$, d and $d+1$ (if present). The value $V_{S2}(j_r, j_c)$ contributes to entry $[j_r, j_c]$ in blocks 0 and 1, and $V_{S1}(j_r, j_c)$ only contributes to the block with index 0. This is how the matrix V_h is assembled in practice, which we will discuss in more detail in the next section. The visualization of matrix entries affected by a given pair of spatial elements and a value of d is shown later (together with other matrices) in Figure 5.

Finally, we need to evaluate the spatial integrals in (15)–(17), which is done using numerical quadrature. For all three cases we perform a standard mapping to a reference triangle $\hat{\gamma}$ (we

use a general function f as the integrand),

$$\int_{\gamma_{j_r}}\int_{\gamma_{j_c}} f(\boldsymbol{x},\boldsymbol{y})\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}\,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} = 4\Delta_{j_r}\Delta_{j_c}\underbrace{\int_{\hat\gamma}\int_{\hat\gamma}\hat f(\hat{\boldsymbol{x}},\hat{\boldsymbol{y}})\,\mathrm{d}\boldsymbol{s}_{\hat{\boldsymbol{y}}}\,\mathrm{d}\boldsymbol{s}_{\hat{\boldsymbol{x}}}}_{=:I},$$
$$= 4\Delta_{j_r}\Delta_{j_c}\int_{\hat\gamma}\int_{\hat\gamma} f(a_{j_r}(\hat{\boldsymbol{x}}),a_{j_c}(\hat{\boldsymbol{y}}))\,\mathrm{d}\boldsymbol{s}_{\hat{\boldsymbol{y}}}\,\mathrm{d}\boldsymbol{s}_{\hat{\boldsymbol{x}}}$$

where Δ_j denotes surface area of element γ_j, and a_{j_r} and a_{j_c} are the affine mappings from the reference element $\hat\gamma$ to the elements γ_{j_r} and γ_{j_c}, respectively.

The integrand in the time-regular contribution V_R is smooth, we can therefore use standard quadrature routines. The same holds for both time-singular contributions V_{S1} and V_{S2} when the test and trial elements are well separated (meaning $\overline{\gamma_{j_r}}\cap\overline{\gamma_{j_c}}=\emptyset$). We use triangle rules, getting an approximation

$$I \approx \sum_{n_1=1}^{N}\sum_{n_2=1}^{N}\hat{\mathrm{w}}_{n_1}\hat{\mathrm{w}}_{n_2}\hat f(\hat{\boldsymbol{x}}_{n_1},\hat{\boldsymbol{x}}_{n_2}),$$

where $\hat{\boldsymbol{x}}_n$ are strategically selected quadrature nodes in the reference triangle and $\hat{\mathrm{w}}_n$ corresponding weights.

For the remaining cases (V_{S1} and V_{S2} with identical elements or elements sharing a node or an edge) we use regularized quadrature based on Duffy substitution. The double integral transforms to a sum over $\widetilde{S}$ simplices of four-dimensional integrals with non-singular integrand. The four-dimensional integral is then approximated using tensor product Gaussian scheme, leading to

$$I \approx \sum_{s=1}^{\widetilde{S}}\sum_{\ell_1=1}^{M}\sum_{\ell_2=1}^{M}\sum_{\ell_3=1}^{M}\sum_{\ell_4=1}^{M}\hat{\mathrm{w}}_{\ell_1}\hat{\mathrm{w}}_{\ell_2}\hat{\mathrm{w}}_{\ell_3}\hat{\mathrm{w}}_{\ell_4}J^s(z_{\ell_1},z_{\ell_2},z_{\ell_3},z_{\ell_4})\hat f(\boldsymbol{F}^s(z_{\ell_1},z_{\ell_2},z_{\ell_3},z_{\ell_4}))$$

with mapping $\boldsymbol{F}^s:\langle 0,1\rangle^4\to S\subset\hat\gamma\times\hat\gamma$, Jacobian $J^s:\langle 0,1\rangle^4\to\mathbb{R}$, and appropriately chosen quadrature points z_ℓ with corresponding weights $\hat{\mathrm{w}}_\ell$. The number of sampling points M, their locations and corresponding weights depends on the relative configuration of the test and trial elements, as well as on chosen quadrature order. For more details regarding the regularized quadrature see [23, 29].

In the end, both quadrature techniques can be viewed as sampling the integrand in strategically selected points and estimating the integral as a weighted sum of the sampled function values,

$$I \approx \sum_{\ell=1}^{L}\mathrm{w}_\ell\hat f(\hat{\boldsymbol{x}}_\ell,\hat{\boldsymbol{y}}_\ell).$$

A summary of when each integration technique is used is shown in Table 2.

Table 2: A summary of quadrature techniques used for different integrals

	relative position of elements			
	disjoint	shared node	shared edge	identical elements
V_{S1}	standard	regularized	regularized	regularized
V_{S2}	standard	regularized	regularized	regularized
V_R	standard	standard	standard	standard

2.3.2 Double layer matrix K_h

The double layer matrix K_h also turns out to have block lower triangular Toeplitz structure with block dimensions $E_t \times E_t$ and size of block $E_s \times N_s$,

$$\mathsf{K}_h = \begin{bmatrix} \mathsf{K}_h^0 & \mathsf{O} & \dots & \mathsf{O} \\ \mathsf{K}_h^1 & \mathsf{K}_h^0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathsf{O} \\ \mathsf{K}_h^{E_t-1} & \dots & \mathsf{K}_h^1 & \mathsf{K}_h^0 \end{bmatrix},$$

with values

$$\begin{aligned} \mathsf{K}_h^d[j_r, j_c] &= \int_{\gamma_{j_r}} \int_{\Gamma_h} K^d(\boldsymbol{x}-\boldsymbol{y}) \varphi_{s,j_c}^1(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} \\ &= \sum_{m=1}^{E_s} \int_{\gamma_{j_r}} \int_{\gamma_m} K^d(\boldsymbol{x}-\boldsymbol{y}) \varphi_{s,j_c}^1(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \end{aligned}$$

where

$$K^d(\boldsymbol{r}) = \begin{cases} \alpha \dfrac{\partial}{\partial \boldsymbol{n}_{\boldsymbol{y}}} \displaystyle\int_0^{h_t} \int_0^t G_\alpha(\boldsymbol{r}, t-\tau) \,\mathrm{d}\tau \,\mathrm{d}t & \text{for } d = 0, \\ \alpha \dfrac{\partial}{\partial \boldsymbol{n}_{\boldsymbol{y}}} \displaystyle\int_{t_d}^{t_{d+1}} \int_0^{h_t} G_\alpha(\boldsymbol{r}, t-\tau) \,\mathrm{d}\tau \,\mathrm{d}t & \text{for } d \in \{1, 2, \dots, E_t - 1\}. \end{cases}$$

We again integrate the temporal integrals analytically (see [30]), resulting in

$$\begin{aligned} \mathsf{K}_h^0[j_r, j_c] &= \widetilde{K}_{S1}(j_r, j_c) + \widetilde{K}_{S2}(j_r, j_c) - \widetilde{K}_R(j_r, j_c, 1), \\ \mathsf{K}_h^1[j_r, j_c] &= -\widetilde{K}_{S2}(j_r, j_c) + 2\widetilde{K}_R(j_r, j_c, 1) - \widetilde{K}_R(j_r, j_c, 2), \\ \mathsf{K}_h^d[j_r, j_c] &= -\widetilde{K}_R(j_r, j_c, d-1) + 2\widetilde{K}_R(j_r, j_c, d) - \widetilde{K}_R(j_r, j_c, d+1) \quad \text{for } d \geq 2, \end{aligned}$$

where

$$\widetilde{K}_{S1}(j_r, j_c) = h_t \sum_{m=1}^{E_s} \int_{\gamma_{j_r}} \int_{\gamma_m} \alpha \frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y}, 0) \varphi_{s,j_c}^1(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{18}$$

$$\widetilde{K}_{S2}(j_r, j_c) = \sum_{m=1}^{E_s} \int_{\gamma_{j_r}} \int_{\gamma_m} \alpha \frac{\partial G_\alpha^{\mathrm{d}\tau\mathrm{d}t}}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y}, 0) \varphi_{s,j_c}^1(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{19}$$

$$\widetilde{K}_R(j_r, j_c, d) = \sum_{m=1}^{E_s} \int_{\gamma_{j_r}} \int_{\gamma_m} \alpha \frac{\partial G_\alpha^{\mathrm{d}\tau\mathrm{d}t}}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x}-\boldsymbol{y}, dh_t) \varphi_{s,j_c}^1(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{20}$$

and

$$\alpha\frac{\partial G_\alpha^{\mathrm{d}\tau\mathrm{d}t}}{\partial \boldsymbol{n_y}}(\boldsymbol{r},\delta) = -\frac{1}{4\pi}\frac{\boldsymbol{r}\cdot\boldsymbol{n_y}}{\|\boldsymbol{r}\|}\left(\left(\frac{1}{2\alpha}-\frac{\delta}{\|\boldsymbol{r}\|^2}\right)\operatorname{erf}\left(\frac{\|\boldsymbol{r}\|}{2\sqrt{\alpha\delta}}\right)+\frac{\sqrt{\delta}}{\|\boldsymbol{r}\|\sqrt{\pi\alpha}}\exp\left(-\frac{\|\boldsymbol{r}\|^2}{4\alpha\delta}\right)\right),$$

$$\alpha\frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n_y}}(\boldsymbol{r},\delta) = \frac{1}{4\pi}\frac{\boldsymbol{r}\cdot\boldsymbol{n_y}}{\|\boldsymbol{r}\|^2}\left(\frac{1}{\|\boldsymbol{r}\|}\operatorname{erf}\left(\frac{\|\boldsymbol{r}\|}{2\sqrt{\alpha\delta}}\right)-\frac{1}{\sqrt{\pi\alpha\delta}}\exp\left(-\frac{\|\boldsymbol{r}\|^2}{4\alpha\delta}\right)\right)$$

with the specially treated limiting cases

$$\lim_{\delta\to 0+}\alpha\frac{\partial G_\alpha^{\mathrm{d}\tau\mathrm{d}t}}{\partial \boldsymbol{n_y}}(\boldsymbol{r},\delta) = -\frac{\boldsymbol{r}\cdot\boldsymbol{n_y}}{8\pi\alpha\|\boldsymbol{r}\|} \quad \text{for } \|\boldsymbol{r}\|>0,$$

$$\lim_{\|\boldsymbol{r}\|\to 0+}\alpha\frac{\partial G_\alpha^{\mathrm{d}\tau\mathrm{d}t}}{\partial \boldsymbol{n_y}} = 0 \quad \text{for } \delta>0,$$

$$\lim_{\delta\to 0+}\alpha\frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n_y}}(\boldsymbol{r},\delta) = \frac{\boldsymbol{r}\cdot\boldsymbol{n_y}}{4\pi\|\boldsymbol{r}\|^3} \quad \text{for } \|\boldsymbol{r}\|>0.$$

The sums in (18)–(20) contain many zeros arising from the function φ^1_{s,j_c}, the only nonzero terms are where the node μ_{j_c} is a vertex of the triangular element γ_m. The spatial double integrals are again evaluated using numerical quadrature.

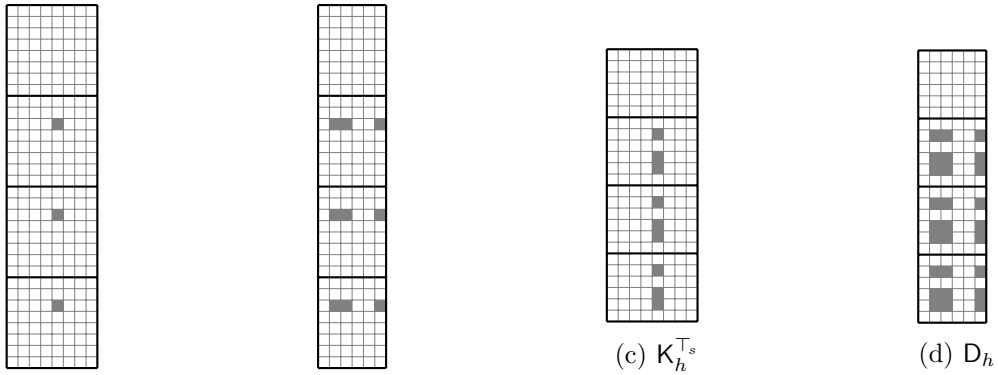
Looking at the evaluation of $\widetilde{K}_R$ and matrix assembly again from a different perspective, every ordered pair of spatial elements γ_{j_r} and γ_m and every $d\in\{1,2,\dots,E_t\}$ contributes to row j_r and three columns corresponding to the vertices of the element γ_m in blocks $d-1$, d and $d+1$, if available. The values of the contribution differ not only by the $\{-1,2,-1\}$ factors for the blocks, but also in the columns. However, the numerical quadrature is performed for the three columns at the same time, reusing the calculated values of the heat kernel antiderivative and only varying the function φ^1_{s,j_c} for the different columns. Similar strategy can be employed for $\widetilde{K}_{S2}$ with blocks 0 and 1, and for $\widetilde{K}_{S1}$ with the $d=0$ block.

2.3.3 Adjoint double layer matrix $\mathsf{K}_h^{\top_s}$

The adjoint double layer matrix again has the block lower triangular Toeplitz structure with block dimensions $E_t\times E_t$ and size of block $N_s\times E_s$. It can be proved, that the matrix $\mathsf{K}_h^{\top_s}$ can be created from K_h by transposing its blocks, therefore

$$\mathsf{K}_h^{\top_s,d}[j_r,j_c] = \mathsf{K}_h^{d}[j_c,j_r].$$

Because of the similarity with the double layer matrix, the adjoint double layer matrix does not need to be explicitly assembled. Whenever we need to use $\mathsf{K}_h^{\top_s}$, we provide K_h and specify a special indicator signaling the blocks of the matrix should be transposed. However, the assembly of the adjoint double layer matrix would be done in a very similar way to the double layer matrix, except the pair of spatial elements contributes in a block to three rows in a single column, as opposed to a single row in three columns.



(a) V_h (b) K_h (c) $\mathsf{K}_h^{\top_s}$ (d) D_h

Figure 5: Matrix entries affected by a pair of spatial elements with indices $j_r = 3$ and $j_c = 5$, and $d = 2$

2.3.4 Hypersingular matrix D_h

Block lower triangular Toeplitz structure also emerges in the case of the hypersingular matrix, with block dimensions $E_t \times E_t$ and size of block $N_s \times N_s$. We will not get into any detail regarding formulas for calculation of the matrix entries. They are, however, calculated in a very similar way to the the previous matrices – the temporal integrals are integrated analytically, while the spatial numerically. Assembling the matrix, every pair of test and trial elements contributes in a block to rows corresponding to the three vertices of the test element and columns corresponding to the vertices of the trial element, thus updating a 3×3 (possibly non-contiguous) submatrix. For details see [30].

Structure of the first block column of the four main boundary element matrices V_h, K_h, $\mathsf{K}_h^{\top_s}$ and D_h is shown in Figure 5. The entries affected in the assembly process by a pair of spatial elements with indices $j_r = 3$ and $j_c = 5$ and temporal element difference $d = 2$ are highlighted.

2.3.5 Mass matrix M_h

The mass matrix M_h represents the identity operator in the boundary integral equations (4) and (5) and in the corresponding variational formulations. It has block dimensions $E_t \times E_t$ with blocks of size $E_s \times N_s$, the same as K_h.

The values of the entries can be calculated as

$$\begin{aligned}
\mathsf{M}_h[i_r, i_c][j_r, j_c] &= \langle \varphi^0_{t,i_c} \varphi^1_{s,j_c}, \varphi^0_{t,i_r} \varphi^0_{s,j_r} \rangle_{\Sigma_h} \\
&= \int_0^T \int_{\Gamma_h} \varphi^0_{t,i_c} \varphi^0_{t,i_r} \varphi^1_{s,j_c} \varphi^0_{s,j_r} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} \,\mathrm{d}t \\
&= \int_0^T \int_{\gamma_{j_r}} \varphi^0_{t,i_c} \varphi^0_{t,i_r} \varphi^1_{s,j_c} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} \,\mathrm{d}t \\
&= \begin{cases} h_t \displaystyle\int_{\gamma_{j_r}} \varphi^1_{s,j_c} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, & i_c = i_r, \\ 0, & i_c \neq i_r. \end{cases}
\end{aligned}$$

Observing that

$$\int_{\gamma_{j_r}} \varphi^1_{s,j_c} \,\mathrm{d}s_{\boldsymbol{x}} = \begin{cases} \frac{1}{3}\Delta_{j_r}, & \mu_{j_c} \text{ is a vertex of } \gamma_{j_r}, \\ 0, & \text{otherwise,} \end{cases}$$

we get

$$\mathsf{M}_h[i_r, i_c][j_r, j_c] = \begin{cases} \frac{1}{3}h_t\Delta_{j_r}, & i_c = i_r \wedge \mu_{j_c} \text{ is a vertex of } \gamma_{j_r}, \\ 0, & \text{otherwise,} \end{cases}$$

where we again denote Δ_j the surface area of spatial element γ_j.

We found the mass matrix M_h to be block diagonal with identical blocks on the diagonal,

$$\mathsf{M}_h = \begin{bmatrix} \mathsf{M}_{h,s} & \mathsf{O} & \dots & \mathsf{O} \\ \mathsf{O} & \mathsf{M}_{h,s} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathsf{O} \\ \mathsf{O} & \dots & \mathsf{O} & \mathsf{M}_{h,s} \end{bmatrix}.$$

The blocks $\mathsf{M}_{h,s}$ are sparse, having exactly three nonzero entries per row in columns corresponding to the vertices of the triangular element represented by the row.

2.4 Evaluation of the solution

After solving the system of equations (12) or (13) we know both the Dirichlet and Neumann data $\boldsymbol{u}$ and $\boldsymbol{w}$. For unification of the two cases, we denote $u_h = g_h$ or $w_h = h_h$ as well as $\boldsymbol{u} = \boldsymbol{g}$ or $\boldsymbol{w} = \boldsymbol{h}$ in the Dirichlet or Neumann boundary condition case, respectively.

To get the value of the solution u in any point $\boldsymbol{x}$ inside the region enclosed by the discretized boundary Γ_h and in any time $t \in \langle 0, T\rangle$, where $t = t_l + \varepsilon = lh_t + \varepsilon$ with $\varepsilon \in \langle 0, h_t)$, we need to evaluate the discretized representation formula (ignoring the zero initial potential term)

$$u(\boldsymbol{x}, t) = \widetilde{V}(w_h)(\boldsymbol{x}, t) - W(u_h)(\boldsymbol{x}, t),$$

where

$$\widetilde{V}(w_h)(\boldsymbol{x}, t) = \int_0^t \int_{\Gamma_h} G_\alpha(\boldsymbol{x} - \boldsymbol{y}, t - \tau) w_h(\boldsymbol{y}, \tau) \,\mathrm{d}s_{\boldsymbol{y}} \,\mathrm{d}\tau,$$

$$W(u_h)(\boldsymbol{x}, t) = \int_0^t \int_{\Gamma_h} \alpha \frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x} - \boldsymbol{y}, t - \tau) u_h(\boldsymbol{y}, \tau) \,\mathrm{d}s_{\boldsymbol{y}} \,\mathrm{d}\tau.$$

Modifying the formulas and integrating analytically over time, we get

$$\begin{aligned} \widetilde{V}(w_h)(\boldsymbol{x}, t) = &\sum_{j=1}^{E_s} \int_{\gamma_j} \sum_{d=0}^{l-1} w_{l-d,j} \left(G_\alpha^{\mathrm{d}\tau}(\boldsymbol{x} - \boldsymbol{y}, dh_t + \varepsilon) - G_\alpha^{\mathrm{d}\tau}(\boldsymbol{x} - \boldsymbol{y}, (d+1)h_t + \varepsilon) \right) \mathrm{d}s_{\boldsymbol{y}} \\ &+ \sum_{j=1}^{E_s} \int_{\gamma_j} w_{l+1,j} \left(G_\alpha^{\mathrm{d}\tau}(\boldsymbol{x} - \boldsymbol{y}, 0) - G_\alpha^{\mathrm{d}\tau}(\boldsymbol{x} - \boldsymbol{y}, \varepsilon) \right) \mathrm{d}s_{\boldsymbol{y}}, \\ W(u_h)(\boldsymbol{x}, t) = &\sum_{j=1}^{N_s} \int_{\Gamma_h} \varphi^1_{s,j}(\boldsymbol{y}) \sum_{d=0}^{k-1} u_{l-d,j} \left(\alpha \frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x} - \boldsymbol{y}, dh_t + \varepsilon) - \alpha \frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x} - \boldsymbol{y}, (d+1)h_t + \varepsilon) \right) \mathrm{d}s_{\boldsymbol{y}} \\ &+ \sum_{j=1}^{N_s} \int_{\Gamma_h} \varphi^1_{s,j}(\boldsymbol{y}) u_{l+1,j} \left(\alpha \frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n}_y}(\boldsymbol{x} - \boldsymbol{y}, 0) - \alpha \frac{\partial G_\alpha^{\mathrm{d}\tau}}{\partial \boldsymbol{n}_{\boldsymbol{y}}}(\boldsymbol{x} - \boldsymbol{y}, \varepsilon) \right) \mathrm{d}s_{\boldsymbol{y}}. \end{aligned}$$

The spatial integrals are again calculated using numerical quadrature, standard techniques suffice due to absence of singularities.

3 Analysis of the current code

The code developed within this thesis is a part of the BESTHEA library (*Boundary Element Solver for The Heat EquAtion* [17]), which aims at efficient parallel solution of problems described in the previous section.

The goal of this section is to give the reader insight into how the library currently functions, with emphasis on the parts of the code which are the target of GPU acceleration. The acceleration of the code itself will be discussed in the next sections.

The library is written in `C++` (specifically the `C++17` standard [6]) and is built using CMake [3] and GNU Make. Intel Math Kernel Library has to be installed on the system, along with any MPI library.

3.1 Overview of the library

The whole library is located inside the `besthea` namespace, which consists of several other namespaces containing classes and structures enabling a user to find numerical solution to the heat equation efficiently and in parallel. This includes classes for managing meshes, matrix and vector types, assembler of the boundary element matrices, evaluator of the potentials and many other. A diagram of the main namespaces, classes and their inheritance hierarchy is shown in Figure 6.

A typical workflow used to find a numerical solution to the heat equation utilizing the BESTHEA library is following.

1. Create space-time mesh,
2. create and populate the boundary condition vector,
3. create and assemble necessary matrices,
4. solve the system,
5. evaluate the solution in points of space-time grid.

An example code demonstrating the numerical solution to a Dirichlet problem with zero initial condition using the library is shown in Listing 1, which we describe in greater detail in the following paragraphs.

The library uses two aliases for basic types – `sc` (for scalar) representing used floating point type, and `lo` (for local ordinal) used for indexing. In this thesis we always use `sc=double` and `lo=long` (64-bit integer on Unix-like 64-bit systems).

The space-time mesh represented by the `uniform_spacetime_tensor_mesh` class is a tensor product of a spatial surface mesh and a uniform discretization of the time interval $(0, T)$. The surface mesh (`triangular_surface_mesh`) can be either created from a tetrahedral volume mesh, or loaded directly from a file as shown in the example on line 31. The spacetime tensor mesh is then created and refined, that is the elements are divided into smaller elements to obtain finer discretization.

The discretized boundary element spaces $X_h^{0,0}$ and $X_h^{0,1}$ are represented by (appropriately templated) class `uniform_spacetime_be_space` and can be created using the space-time mesh (lines 36 and 37). Using their `L2_projection` method (as shown on line 41), one can project a user-defined boundary condition function (we use `bc_dir_func` defined on lines 8–20) onto the

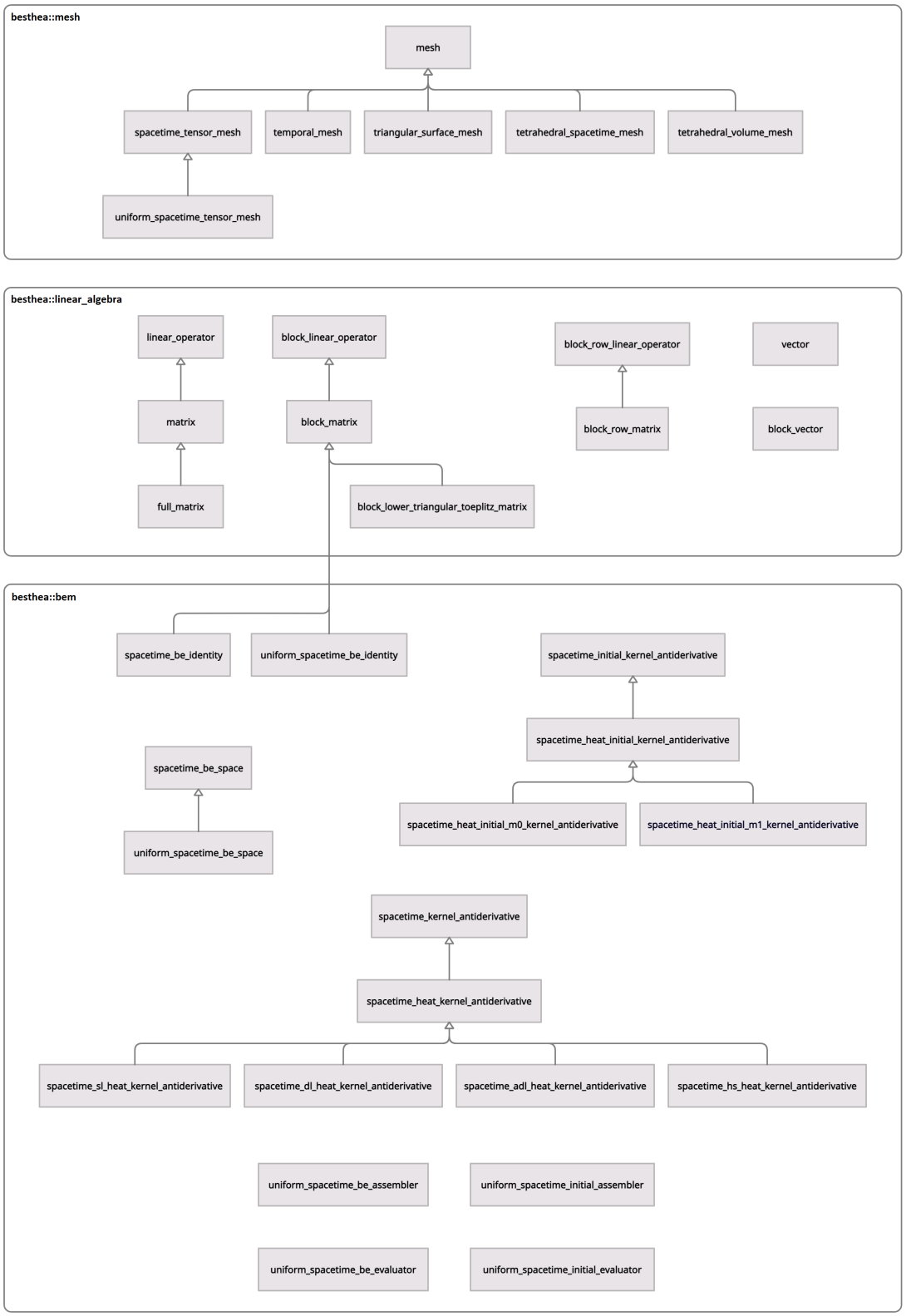


Figure 6: Diagram of the main classes and namespaces in BESTHEA library

```cpp
#include <string>
#include <besthea/besthea.h>

using namespace besthea::mesh;
using namespace besthea::bem;
using namespace besthea::linear_algebra;

sc bc_dir_func( sc x1, sc x2, sc x3, const coordinates< 3 > &, sc t ) {
    constexpr std::array< sc, 3 > _y{ 0.0, 0.0, 1.5 };
    sc alpha = 0.5;

    sc norm2 = ( x1 - _y[ 0 ] ) * ( x1 - _y[ 0 ] )
               + ( x2 - _y[ 1 ] ) * ( x2 - _y[ 1 ] )
               + ( x3 - _y[ 2 ] ) * ( x3 - _y[ 2 ] );

    sc value = std::pow( 4.0 * M_PI * alpha * t, -1.5 )
               * std::exp( -norm2 / ( 4.0 * alpha * t ) );

    return value;
}

int main()
{
    sc alpha = 0.5;

    // load and create mesh
    std::string mesh_file = "path/to/mesh/cube_surf.txt";
    lo n_timesteps = 8;
    sc end_time = 1.0;
    triangular_surface_mesh space_mesh;
    space_mesh.load(mesh_file);
    uniform_spacetime_tensor_mesh spacetime_mesh(space_mesh, end_time, n_timesteps);
    spacetime_mesh.refine(1);

    // create BE spaces
    uniform_spacetime_be_space<basis_tri_p0> space_p0(spacetime_mesh);
    uniform_spacetime_be_space<basis_tri_p1> space_p1(spacetime_mesh);

    // project the boundary condition onto the BE space
    block_vector bc_dir;
    space_p1.L2_projection( bc_dir_func, bc_dir );

    // create and assemble single layer matrix V
    block_lower_triangular_toeplitz_matrix V;
    spacetime_heat_sl_kernel_antiderivative kernel_v( alpha );
    uniform_spacetime_be_assembler assembler_v(kernel_v, space_p0, space_p0);
    assembler_v.assemble(V);

    // create and assemble double layer matrix K
    block_lower_triangular_toeplitz_matrix K;
    spacetime_heat_dl_kernel_antiderivative kernel_k( alpha );
    uniform_spacetime_be_assembler assembler_k(kernel_k, space_p0, space_p1);
    assembler_k.assemble(K);

    // create and assemble mass matrix M
    uniform_spacetime_be_identity M(space_p0, space_p1);
    M.assemble();

    // create and assemble right hand side vector
    block_vector rhs( V.get_block_dim(), V.get_n_rows() );
    M.apply(bc_dir, rhs, false, 0.5, 0.0);
    K.apply(bc_dir, rhs, false, 1.0, 1.0);

    // solve the system
    block_vector sol_neu( V.get_block_dim(), V.get_n_columns() );
    sc rel_error = 1e-6;
    lo n_iters = 1000;
    V.mkl_fgmres_solve(rhs, sol_neu, rel_error, n_iters);

    // load volume mesh
    std::string grid_file = "path/to/mesh/cube_vol.txt";
    tetrahedral_volume_mesh vol_mesh;
    vol_mesh.load(grid_file);

    // evaluate single layer potential
    block_vector slp;
    uniform_spacetime_be_evaluator evaluator_v(kernel_v, space_p0);
    evaluator_v.evaluate(vol_mesh.get_nodes(), sol_neu, slp);

    // evaluate double layer potential
    block_vector dlp;
    uniform_spacetime_be_evaluator evaluator_k(kernel_k, space_p1);
    evaluator_k.evaluate(vol_mesh.get_nodes(), bc_dir, dlp);

    // combine the potentials to get the final solution
    block_vector solution_grid(slp);
    solution_grid.add(dlp, -1.0);

    // do something with the solution ...

    return 0;
}
```

Listing 1: Solution of the Dirichlet problem using the BESTHEA library

appropriate discrete space, thus populating the boundary condition vector, which is an instance of the `block_vector` class.

We then create and assemble all the necessary matrices required for solving the problem (lines 44–57). The main boundary element matrices are all block lower triangular with block Toeplitz structure, thus are represented by the `block_lower_triangular_toeplitz_matrix` class. To get an assembled main boundary element matrix, we first create an empty matrix, create instance of an assembler class `uniform_spacetime_be_assembler`, and finally call the `assemble` method on the assembler with the matrix as a parameter to fill it with values. Creation of the mass matrix is simpler, we only need to instantiate the class `uniform_spacetime_be_identity` and call its method `assemble`.

Since $\mathsf{K}_h^{\top_s}$ can be obtained from K_h by transposing its blocks, only K_h is actually needed. Whenever we need to use $\mathsf{K}_h^{\top_s}$, we provide K_h and specify a special indicator marking that the blocks of the matrix should be transposed. Analogous approach can be used with the mass matrices M_h and $\mathsf{M}_h^{\top}$.

After assembling the right-hand side vector on lines 60–62 using the `apply` method on the matrices to perform matrix-vector multiplication, the system of equations is ready to be solved. This is usually done by calling the `mkl_fgmres_solve` method on the system matrix, as shown on line 68. This method uses FGMRES algorithm (flexible generalized minimal residual method [22]) implemented in the Intel Math Kernel Library (MKL).

After the system is solved, both the Dirichlet and Neumann data $\boldsymbol{u}$ and $\boldsymbol{w}$ are known and can be used to calculate values of the solution in points of space-time grid using the representation formula. We load a volume mesh (see line 73) and create appropriate instances of the `uniform_spacetime_be_evaluator` class for evaluating the single and double layer potential (lines 77 and 82). Using their `evaluate` method the values of the potentials are calculated in nodes of the volume mesh in all timesteps, as shown on lines 78 and 83. Finally, the single and double layer potentials are combined to fill the block vector with values of the solution in the space-time grid on lines 86 and 87.

3.2 Current implementation

In this subsection we go through the current implementation of the library. We focus more on the algorithmic side of the implementation and will not discuss every line of code in detail. We focus mainly on the assembly of the main boundary element matrices and their multiplication with vectors, since this is the target of GPU acceleration.

3.2.1 Vector and matrix classes

All the vector and matrix types are located in namespace `besthea::linear_algebra`. The class used for (mathematical) vectors is `vector`, which uses `std::vector`[3] to store its data. Block vector is represented by the `block_vector` class, in which every block is stored as a separate `vector` in `std::vector`. This means the data are not all linear in memory, only the individual blocks themselves. In some scenarios it is useful to view a block vector as a matrix, where each column represents one block of the block vector. Because of this, the vector of vectors data layout can be sometimes impractical due to incompatibility with BLAS (Basic Linear Algebra

[3]The `vector` class from the `C++` Standard Template Library

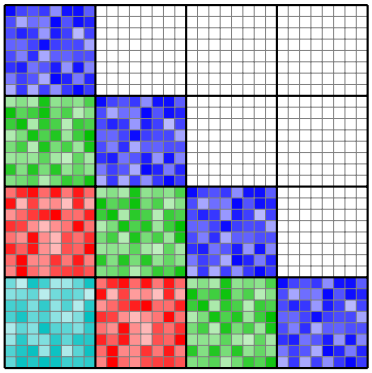


Figure 7: Structure of block lower triangular Toeplitz matrix

Subprograms) data layout philosophy, which is to have all the data of a matrix stored in one contiguous chunk of memory.

The `block_lower_triangular_toeplitz_matrix` class stores the assembled main boundary element matrix. Taking advantage of the block lower triangular Toeplitz structure (visualized in Figure 7), only the first block column needs to be stored. The individual blocks are stored in instances of the `full_matrix` class. This class represents a full matrix and utilizes `std::vector` to store all its data in one contiguous chunk of memory in column-major order. Memory is allocated for the exact number of entries needed, there are no paddings in columns (only a padding at the end of the memory block), therefore the leading dimension is equal to the number of rows in the full matrix.

The block lower triangular Toeplitz matrix-vector multiplication loops through the E_t distinct blocks and for each of them loops through all of its $E_t - d$ possible placements in the corresponding block d-subdiagonal. For every placement, a classic matrix-vector multiplication is performed with corresponding blocks of the vectors `x` and `y`. Implementation of the `apply` method is show in Listing 2.

If we viewed the block vector as a matrix, we could notice, that the loop iterating through all blocks on a block subdiagonal and multiplying them with blocks of the block vector could be replaced with a single matrix-matrix multiplication. The first of the matrices would be the block of the matrix, while the second an appropriate submatrix of the matrix representing the block vector. This would allow for usage of a matrix-matrix multiplication algorithm with a more favorable time complexity and provide better opportunity for parallelization.

3.2.2 Assemblers for the main boundary element matrices

The `uniform_spacetime_be_assembler` class resides in the `besthea::bem` namespace. It is a template class with three template parameters. First of them is a class evaluating heat kernel antiderivatives, the other two represent the test and trial spaces (piecewise constant or piecewise linear in space, always piecewise constant in time). For each of the main boundary element matrices a specifically templated assembler has to be used. The template arguments however do not need to be explicitly specified and are inferred by the compiler from types of constructor arguments. The correct way of creating assemblers for the main boundary element matrices is shown in code Listing 3 (note that $\mathsf{K}_h^{\top_s}$ is not shown since it is not needed to be explicitly assembled). The constructor of the assembler class has two additional optional parameters

```
void besthea::linear_algebra::block_lower_triangular_toeplitz_matrix::apply(
    const block_vector & x, block_vector & y,
    bool trans, sc alpha, sc beta ) const {

    const full_matrix * m;
    const vector * subx;
    vector * suby;

    sc block_beta = beta;
    for ( lo diag = 0; diag < _block_dim; ++diag ) {
        m = &( _data[ diag ] );
        for ( lo block = 0; block < _block_dim - diag; ++block ) {
            subx = &( x.get_block( block ) );
            suby = &( y.get_block( block + diag ) );
            m->apply( *subx, *suby, trans, alpha, block_beta );
        }
        block_beta = 1.0;
    }
}
```

Listing 2: Block lower triangular Toeplitz matrix apply method

```
uniform_spacetime_be_space<basis_tri_p0> space_p0(spacetime_mesh);
uniform_spacetime_be_space<basis_tri_p1> space_p1(spacetime_mesh);

spacetime_heat_sl_kernel_antiderivative kernel_v(alpha);
spacetime_heat_dl_kernel_antiderivative kernel_k(alpha);
spacetime_heat_hs_kernel_antiderivative kernel_d(alpha);

uniform_spacetime_be_assembler assembler_v(kernel_v, space_p0, space_p0);
uniform_spacetime_be_assembler assembler_k(kernel_k, space_p0, space_p1);
uniform_spacetime_be_assembler assembler_d(kernel_d, space_p1, space_p1);
```

Listing 3: Creation of main boundary element matrix assemblers

specifying the order of numerical quadrature used for evaluating the spatial integrals in the time-singular and time-regular contributions.

Inside the assembler class, there is an auxiliary structure named `quadrature_wrapper`, which wraps the necessary data needed for numerical calculation of integrals to make their privatization for OpenMP threads simpler. It contains coordinates of quadrature nodes in a reference triangle for both test and trial elements for all four of their possible relative configurations (disjoint elements, shared node, shared edge, identical element), and the corresponding quadrature weights. There is also storage for coordinates of the quadrature nodes mapped to the specific elements.

The main boundary element matrices are assembled using the `assemble` method taking as a parameter the matrix object to be filled with values. There are template specializations of the method for three of the four main boundary element matrices ($\mathsf{K}_h^{\top_s}$ is not needed) and a general implementation of the `assemble` method for other valid templatizations of the assembler class.

All specialized implementations of the `assemble` method operate in a similar way. The matrix (provided as a parameter) is first resized to the required dimensions. Then an OpenMP parallel block begins and each thread creates its own `quadrature_wrapper` instance and initializes it using the `init_quadrature` method, which populates it with quadrature node coordinates in a

```
for ( lo delta = 0; delta <= n_timesteps; ++delta ) {
    for( lo i_test = 0; i_test < n_elements; ++i_test ) {
        for( lo i_trial; i_trial < n_elements; ++i_trial ) {

            triangles_to_geometry(i_test, i_trial, my_quadrature);

            if ( delta == 0 ) {
                value = quadrature_V_S1(my_quadrature, ...);
                value *= test_area * trial_area * timestep;

                global_matrix.add( 0, i_test, i_trial, value );
            }

            if ( delta == 0 ) {
                value = quadrature_V_S2(my_quadrature, ...);
            } else {
                if ( i_test != i_trial ) {
                    value = quadrature_V_R_regular(my_quadrature, ...);
                } else {
                    value = quadrature_V_R_singular(my_quadrature, ...);
                }
            }
            value *= test_area * trial_area;

            if ( delta > 0 ) {
                global_matrix.add( delta - 1, i_test, i_trial, -value );
                if ( delta < n_timesteps ) {
                    global_matrix.add( delta, i_test, i_trial, 2.0 * value );
                }
            } else {
                global_matrix.add( 0, i_test, i_trial, value );
            }
            if ( delta < n_timesteps - 1 ) {
                global_matrix.add( delta + 1, i_test, i_trial, -value );
            }

        }
    }
}
```

Listing 4: Assembly of single layer matrix V_h (simplified)

reference element and corresponding weights.

Then the process of filling the matrix with values begins. There are three nested loops iterating through all temporal element differences d, test and trial spatial elements. For every combination of them, several entries of the matrix are updated. The specifics of this depend on the matrix (templatization of the assembler), which we will now discuss. The parallelization is employed on the loop iterating through test elements.

Assembly of single layer matrix V_h

The single layer matrix is the simplest one to assemble. A simplified version of the assembly method is shown in Listing 4. We will also refer to the Section 2.3.1 regarding the calculation of matrix entries. In the code, the variable `delta` represents d, the temporal element index difference, and `i_test` and `i_trial` are the indices of the test and trial elements, previously denoted by j_r and j_c.

For every iteration of the innermost loop we first map the quadrature node coordinates from the reference element to the actual test and trial elements using the `triangles_to_geometry` method. The mapped coordinates are then stored in a variable of type `quadrature_wrapper` named `my_quadrature`.

If the condition `delta == 0` holds, we contribute the first time-singular contribution V_{S1} to the main block diagonal, as seen on lines 7–12. Then, if the same condition is satisfied, we calculate the value of V_{S2} on lines 15 and 23 and contribute it to the first two blocks on lines 31 and 34. Otherwise, if `delta > 0`, on lines 16–23 we calculate the value of the time-regular contribution V_R. The calculation is split to two cases – for identical elements (singular) and other configurations (regular). The only difference between them is, that in the singular case additional checks are performed to take care of possible singularities occurring in the evaluation of the heat kernel antiderivative, so that the appropriate formula is used. In the regular case singularities never occur, therefore the checks are not needed. The value is then contributed to the three blocks (if present) on lines 26, 28 and 34.

Assembly of double layer matrix K_h

Assembly of the double layer matrix is similar to the single layer matrix, with the difference that in each iteration of the innermost loop three entries are updated in each of the three blocks `delta-1`, `delta` and `delta+1`. The updated entries are in a row corresponding to the test element and in columns corresponding to the three vertices of the trial element triangle.

As mentioned in Section 2.3.2, the three values of the contribution are calculated in a single loop, reusing the calculated values of the heat kernel antiderivative and only varying the function φ^1_{s,j_c} for the different columns.

As mentioned in section 2.3.3, the adjoint double layer matrix can be obtained from K_h by transposing its blocks.

Assembly of hypersingular matrix D_h

Each iteration of the innermost loop in the hypersingular matrix assembler updates a (possibly non-contiguous) 3×3 submatrix in the three blocks of the matrix. The rows correspond to the vertex indices of the test element and the columns to vertex indices of the trial element. The quadrature is again performed for all 9 values in a single loop, reusing the values of the heat kernel antiderivative and only varying the functions φ^1_{s,j_r} and φ^1_{s,j_c}.

3.2.3 Solving the system

The system of linear equations is solved using the FGMRES method, which is implemented in the `mkl_fgmres_solve` method in the `block_linear_operator` class and utilizes the FGMRES algorithm from Intel MKL. It does not perform matrix-vector multiplication by itself, for this purpose the control is returned to the user, setting a flag indicating the matrix-vector multiplication should be performed, along with other flags, e.g. marking the position of the vectors in memory. The user is then responsible for performing the operation with the correct data and returning control to the solver.

4 Using GPUs to accelerate scientific codes

CPUs (central processing units) are suitable for a wide range of workloads, having usually a few tens of cores (16–32 on high-end machines), which are focused more on sequential performance. Today's GPUs (graphics processing unit) on the other hand consist of a large number (thousands) of less performant more energy-efficient cores, which are more specialized and the focus is on efficient parallelism. In Table 3 one can find the performance (in double precision) of some of the NVIDIA Tesla accelerators used in HPC [28].

The GPUs have been in the past used mainly to handle display output and rendering, but have since transferred to devices capable of general purpose data processing. Compared to CPUs, they offer higher performance and better energy efficiency arising from massive parallelism. They are, however, not suitable for all types of workloads, as high degree of parallelism is required for acceptable performance. GPUs specialize and devote more transistor to the data processing itself rather than caching and flow control, as illustrated in Figure 8.

Today there are two main players on the dedicated GPU market, Nvidia and AMD. The HPC GPU market is currently absolutely dominated by Nvidia, with AMD having only a single system using their GPUs in the TOP500 list [15]. This is, however, about to change, as the new European supercomputer LUMI (planned to be put in operation in late 2021) will gain most of its computing power from AMD GPUs [12].

Due to its high computational intensity and potential for parallelization, BEM is well suited for acceleration using GPUs. In one of the early works [24] authors use the on-the-fly approach to accelerate the boundary element method for the Helmholtz equation. More recently, the focus has been on acceleration of the fast BEM techniques, such as adaptive cross approximation [5,25] or fast multipole method [26]. An example of an open-source GPU-accelerated library of BEM-based solvers for the Laplace, Helmholtz, and Maxwell problems is Bempp-cl [2]. To the best of our knowledge, there is currently no publicly available software supporting GPU-accelerated implementation of the space-time BEM for the heat equation.

Let us only briefly describe the basics of GPU programming. A reader interested in more details should consult, e.g., [9, 16].

4.1 GPU programming

There are multiple techniques to use the GPU for general purpose computing. The most high-level is to use programs and libraries which are able to utilize the GPU for computing, like MATLAB or TensorFlow. The middle-level approach is to use directives, hinting the compiler to generate code executable on GPUs. This includes mainly OpenACC [19] and OpenMP [20]. The low-level approach is to write the code running on the GPU ourselves using specialized tools

Table 3: Performance of NVIDIA Tesla accelerators

Model name	Release	Processing power [GFlops]
NVIDIA Tesla P100	2016	5304
NVIDIA Tesla V100	2017	7450
NVIDIA Tesla A100	2020	9700

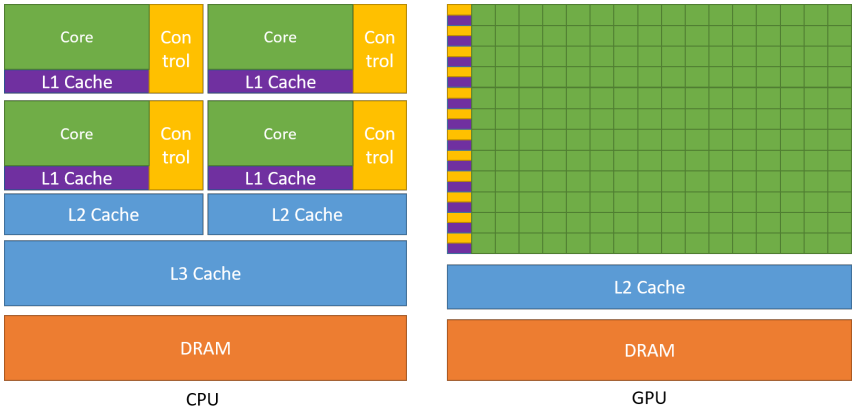


Figure 8: Comparison of CPU and GPU architecture. Image taken from the CUDA Programming guide [16]

```
1  __global__ void my_daxpy(float *x, float *y, float alpha)
2  {
3      int index = blockIdx.x * blockDim.x + threadIdx.x;
4
5      if(index < N)
6          y[index] += alpha * x[index];
7  }
8
9  void main()
10 {
11     // ...
12     my_daxpy<<< 8, 128 >>>(d_x, d_y, 3.14);
13     // ...
14 }
```

Listing 5: An example of CUDA kernel function

and language extensions. Examples of this are CUDA [16], OpenCL [8] and HIP [1]. For the acceleration of the BESTHEA library we use CUDA, which we will now briefly describe.

The largest unit of execution in the CUDA programming model is a kernel function defined using the `__global__` specifier, as can be seen on the first line of an example code in Listing 5. The kernel can be launched using a triple angle bracket syntax, as demonstrated on line 12. The kernel definition, declaration and launch have to be all located in a `*.cu` source file, which can be compiled using `nvcc` compiler. The `*.cu` sources can contain any standard `C++` code, `nvcc` is just a wrapper only taking care of the CUDA-related syntax.

Launching the kernel causes the kernel function to be executed on the GPU device as many times as specified using the triple angle brackets syntax. The first number is the number of blocks, the second is the number of threads per block. The threads are organized into a 2-level hierarchy, the top level being the whole grid composed of a number of blocks (threadblocks), where each threadblock contains several threads. The grid and threadblock can be up to 3-dimensional, their number then has to be specified with value of type `dim3`. An example of a 2-dimensional thread hierarchy is visualized in Figure 9. Inside the kernel every thread in every block gets a unique value of `blockIdx` and `threadIdx`, which are of type `dim3`, identifying the thread in the hierarchy.

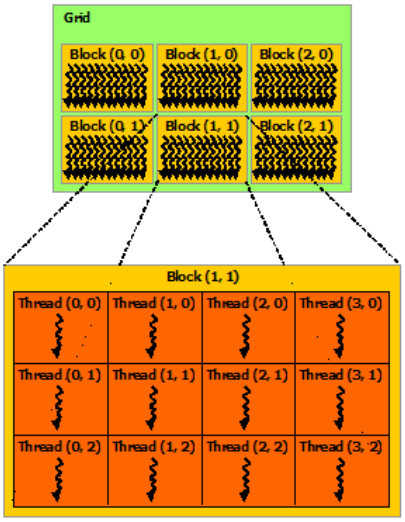


Figure 9: Hierarchy of threads in CUDA. Image taken from the CUDA Programming guide [16]

It is common to call the GPU also as the device, the CPU is usually called the host. These terms are also used to denote memory locations and functions in a source code.

In the `*.cu` files there can be three types of functions – host, device and kernel functions. Host functions (marked `__host__`) are executed on the CPU and can only be called from the host code. Device functions (specified by `__device__`) are executed on the GPU and can only be called from the device code. Kernel functions are callable from both the host and the device, are denoted with `__global__` and run on the device. If not explicitly specified, the function is a host function.

Execution of kernel

The GPU has a number of streaming multiprocessors (SM). Executing the kernel, each threadblock is scheduled for execution to one of the SMs. There the threadblocks are split into warps, each containing 32 threads. Warp is the unit of scheduling on the SM. All threads in a warp execute simultaneously, utilizing the SIMT (single instruction multiple threads) paradigm. If a kernel contains an `if` statement and the results of the condition are not the same for all threads in a warp, both the `true` and `false` branches are executed by the warp with the threads appropriately masked. This implies that high degree of branching is very inefficient.

Types of memory

The main memory on the GPU is the global memory. It can be allocated and freed from the host code, as well as copied to and from. However, we need to distinguish whether a pointer points to host or device memory. Using unified memory (which is still global memory, just with different approach) we do not need to care about where the pointer points, accesses from both the host and device are valid. Modifications to the unified memory are automatically synchronized between the host and device memory.

Each threadblock can allocate a certain amount of shared memory (in orders around 64 KB) using the `__shared__` keyword in variable declaration. It can be accessed by all threads in

the threadblock and is therefore useful for communication between threads. Shared memory is a managed L1 cache and can therefore be accessed with much lower latency compared to global memory. As all threads of a threadblock can access the same shared memory, there exist synchronization mechanisms, such as `__syncthreads()` function representing a barrier within a threadblock. Atomic operations are also supported.

Another type of memory is constant memory. It is a special type of global memory which cannot be modified during kernel execution and has its own constant cache. The constant memory is useful when the same value is to be read by all threads within a warp at the same time. An example of a variable that would be suitable for constant memory is the `alpha` scalar in the `my_daxpy` kernel function example.

For completeness we also mention texture memory, which is yet another type of global memory with its own cache. It is an up to 3-dimensional array, allows addressing with floating point values and implements automatic handling for over-the-bound reads. The texture memory is optimized for 2D spatial locality.

5 Acceleration of the code

One of the disadvantages of the current approach is that the blocks of the four main boundary element matrices are full, therefore consume large amounts of memory. The matrix V_h requires roughly `sizeof(double)`$E_t E_s^2$ bytes of memory. On a machine with 192 GB of memory, the largest mesh for which we are able to numerically solve the heat equation using the previously described implementation of the library has only around 8500 spatial and 350 temporal elements.

There are currently two methods being developed to overcome the memory problem. The first of them is to parallelize the algorithm in distributed memory and use the parabolic fast multipole method (pFMM) to approximate far-field entries [14, 27]. The second approach is not to assemble and store the matrices in memory at all, but to calculate the matrix entries during matrix-vector multiplication on the fly as they are needed. The penalty of calculating all the matrix entries during each multiplication is very large, but using the massive computational power of today's GPUs should partly negate this issue.

The core objective of this thesis is therefore to implement an algorithm that performs on-the-fly matrix-vector multiplication using GPUs, where the matrices arise from the space-time boundary element method for the heat equation.

We first implemented the algorithm for the CPU and used it as a base for the GPU-accelerated code. The algorithm is implemented for the four main boundary element matrices V_h, K_h, $\mathsf{K}_h^{\top_s}$, and D_h. The mass matrices are sparse, therefore do not require the attention. The classes associated with the on-the-fly algorithm are located in the `besthea::bem::onthefly` namespace. All the source code is available in the attachment (see Appendix A).

Matrix-vector multiplication

It is very common in libraries implementing the matrix-vector multiplication to perform a generalized operation in the form `y = alpha*A*x + beta*y` with `alpha` and `beta` being scalars. E.g., if we desired to perform `y = A*x`, we would choose `alpha=1` and `beta=0`. Additional parameter denoting whether the matrix `A` should be transposed is also usually present. In the BESTHEA library this generalized operation is implemented in methods named `apply`.

Since the goal is to implement a (more complicated version of) matrix-vector multiplication, it is very helpful to view it from several different perspectives. The one we found the most useful is visualized in Figure 10. We place the vector `x` on top of the matrix, while the vector `y` is put to the right of it. Each entry of the matrix has a contribution to the vector `y`. The value of this contribution is the value of the matrix entry itself multiplied with an entry of the vector `x` with equal column index. The result is added to an entry of vector `y` with equal row index.

5.1 CPU on-the-fly matrix

The `uniform_spacetime_be_matrix_onthefly_cpu` class represents a main boundary element on-the-fly matrix. It has the same template parameters as the assemblers described in Section 3.2.2 – a class evaluating the heat kernel antiderivatives and classes representing the test and trial spaces. Parameters of the constructor are the same as well. Creation of the four main boundary element on-the-fly matrices is shown in Listing 6. Notice the matrix $\mathsf{K}_h^{\top_s}$ has also its specialization, since we do not support transpositions. A more elaborate examples of usage of the on-the-fly matrices can be found in the attachment (see Appendix A).

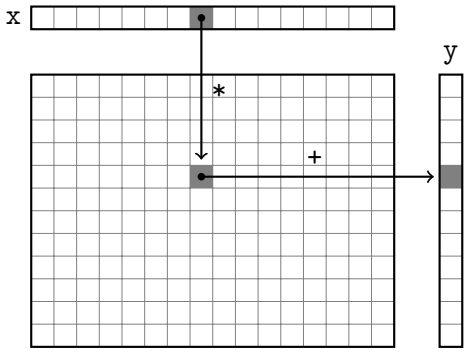


Figure 10: Visualization of matrix-vector multiplication

```
// ...

spacetime_heat_sl_kernel_antiderivative  kernel_v ( heat_capacity_constant );
spacetime_heat_dl_kernel_antiderivative  kernel_k ( heat_capacity_constant );
spacetime_heat_adl_kernel_antiderivative kernel_kt( heat_capacity_constant );
spacetime_heat_hs_kernel_antiderivative  kernel_d ( heat_capacity_constant );

uniform_spacetime_be_matrix_onthefly_cpu V ( kernel_v,  space_p0, space_p0 );
uniform_spacetime_be_matrix_onthefly_cpu K ( kernel_k,  space_p0, space_p1 );
uniform_spacetime_be_matrix_onthefly_cpu Kt( kernel_kt, space_p1, space_p0 );
uniform_spacetime_be_matrix_onthefly_cpu D ( kernel_d,  space_p1, space_p1 );

K.apply( ... );
V.mkl_fgmres_solve( ... );
```

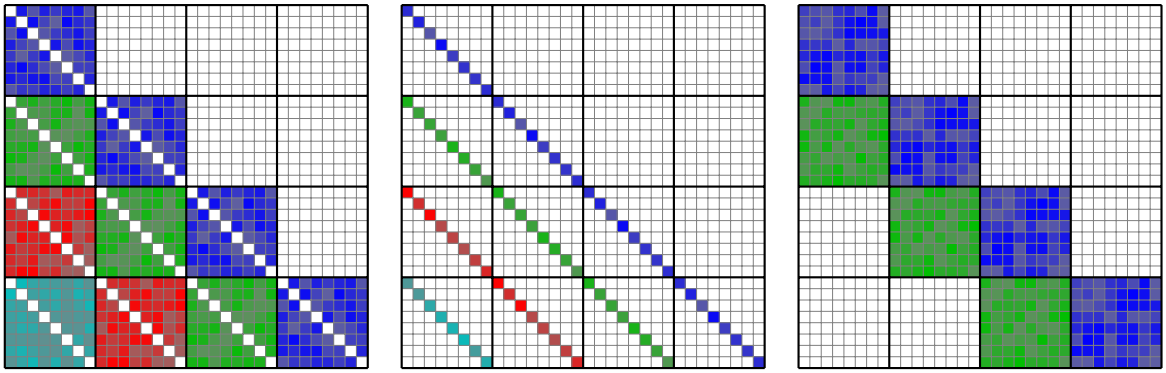
Listing 6: CPU on-the-fly matrix creation

The `uniform_spacetime_be_matrix_onthefly_cpu` class inherits from the `block_matrix` and overrides its `apply` method. After the matrix instance has been created, the `apply` method can be called right away, no assembly is needed. The same holds for the `mkl_fgmres_solve` method, which internally utilizes the `apply` method.

In the `uniform_spacetime_be_matrix_onthefly_cpu` class we use two structures with functionality similar to the `quadrature_wrapper` mentioned in Section 3.2.2. The first of them, `quadrature_reference`, stores quadrature node coordinates in the reference triangle with their corresponding weights. After initialization in the constructor it stays constant and can therefore be accessed from multiple threads. The second structure, `quadrature_nodes`, stores the node coordinates mapped to a specific element. Because they change with every element, each thread will have two private instances of this structure – for the coordinates mapped to the test and trial elements.

5.1.1 Overview of the apply method

For the purpose of the on-the-fly matrix-vector multiplication we split the matrix to a sum of three components. First of them contains all fully regular local contributions, the second all the time-regular space-singular local contributions and the third contains all the time-singular local contributions. This is useful, because local contributions in those three components are calculated differently. Effectively we are taking several `if` statements completely outside of a



(a) Fully regular (b) Time-regular space-singular (c) Time-singular

Figure 11: Matrix entries corresponding to components of a single layer matrix

triple-nested loop. Matrix entries corresponding to each of the three components of a single layer matrix V_h are visualized in Figure 11.

The `apply` method first scales the vector `y` and then sequentially applies the three components, adding the partial results to the vector `y`. For application of each component we created a separate method in the `uniform_spacetime_be_matrix_onthefly_cpu` class. Because the components are different for different matrices, the three methods have specialized implementations for each of the four main boundary element matrices, resulting in a total of 12 implementations of the methods.

The on-the-fly `apply` method works in a similar way as the assemblers do, but instead of adding the calculated local contributions to the matrix, they are (almost) immediately multiplied with the corresponding entries from the input vector `x` and added to the result vector `y`. The code calculating the values of the local contributions is located in separate methods to split the complexity of the code and make it more readable.

5.1.2 Calculating local contributions

Each of the four types of local contributions (fully regular, time-regular space-singular, first time-singular and second time-singular) has an associated method for calculating the local contribution values for a given triplet consisting of test and trial elements and temporal element difference. A given triplet contributes differently to each of the four main boundary element matrices, therefore each such method has a specialized implementation for all four main boundary element matrices. This results in having a total of 16 implementations of the methods calculating the local contributions. The contribution type is specified by the method name, the matrix is specified by template parameters of the class. A simplified version of the method calculating fully regular contribution to the single layer matrix is shown in Listing 7.

The method takes as a parameter the indices of the test and trial elements along with the temporal element difference `delta`, and two instances of the `quadrature_nodes` structure containing quadrature nodes mapped to the test and trial elements. The method performs the numerical quadrature by looping through all the quadrature nodes, evaluating the heat kernel antiderivative value at those points and taking their weighted sum. The results are returned via a pointer also provided as a parameter. For the matrix V_h it returns only a single value, for K_h and $\mathsf{K}_h^{\top_s}$ it calculates and returns three values of the local contributions (corresponding to the

```
template<>
void besthea::bem::onthefly::uniform_spacetime_be_matrix_onthefly_cpu<
    besthea::bem::spacetime_heat_sl_kernel_antiderivative,
    besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >,
    besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 > >::
    get_local_contributions_treg_sreg(sc * values_out,
        lo delta, lo i_test, lo i_trial,
        const quadrature_nodes & qn_tst,
        const quadrature_nodes & qn_trl) const {

    sc timestep = _test_space->get_mesh( )->get_timestep( );
    sc ttau = timestep * delta;
    sc test_area = _test_space->get_mesh( )->spatial_area( i_test );
    sc trial_area = _trial_space->get_mesh( )->spatial_area( i_trial );

    const sc * w = quadr_reference._w[0].data( );
    lo quadr_size = quadr_reference._sizes[0];
    sc value = 0;

    for ( lo i_quad = 0; i_quad < quadr_size; ++i_quad ) {
        value += _kernel->anti_tau_anti_t_regular_in_time_regular_in_space(
            qn_tst.xs[ i_quad ] - qn_trl.xs[ i_quad ],
            qn_tst.ys[ i_quad ] - qn_trl.ys[ i_quad ],
            qn_tst.zs[ i_quad ] - qn_trl.zs[ i_quad ],
            nullptr, nullptr, ttau
        ) * w[ i_quad ];
    }

    sc multiplier = test_area * trial_area;
    *values_out = value * multiplier;
    return;
}
```

Listing 7: Method calculating fully regular local contribution to the single layer matrix

three columns or rows in the matrix block), and for the matrix D_h a 3×3 matrix of values is returned as an array of size 9 containing the values in row-major order.

5.1.3 Applying the components

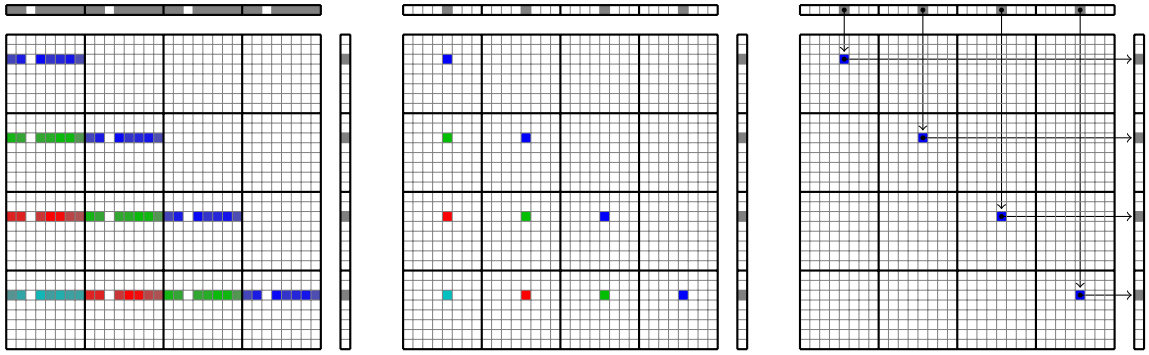
As previously mentioned, we have a total of 12 implementation of the methods performing the apply operation on the three components for all four main boundary element matrices. Similarly to the local contributions, the component is specified by the name of the method, the matrix by templatization of the class.

In the following paragraphs we will focus mainly on explaining the application of the fully regular component of the single layer matrix. The method performing the apply operation is shown in Listing 8. Its parameters are the vectors `x` and `y` along with a scalar `alpha` explained before. The last two parameters specifying a range of test elements that should be considered are specific only to the fully regular component and do not appear in the other components. The reason to this will be explained later in Section 5.2.6, for now we can assume the range covers all test elements.

Right at the beginning we start an OpenMP parallel block in which each thread creates two private instances of the `quadrature_nodes` structure for storing the mapped quadrature node coordinates. Several other private variables are declared here. Then we loop through the test

```
template<>
void besthea::bem::onthefly::uniform_spacetime_be_matrix_onthefly_cpu<
    besthea::bem::spacetime_heat_sl_kernel_antiderivative,
    besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >,
    besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 > >::
    apply_cpu_treg_sreg( const block_vector_type & x_perm,
        block_vector_type & y_perm, sc alpha,
        lo tst_elem_start, lo tst_elem_end ) const {

    lo trl_spelems_count = _trial_space->get_mesh()->get_n_spatial_elements();
    lo n_blocks = _block_dim;

#pragma omp parallel
    {
        quadrature_nodes quadr_nodes_tst(quadr_reference._sizes[0]);
        quadrature_nodes quadr_nodes_trl(quadr_reference._sizes[0]);
        sc val_prev, val_curr, val_next, matrix_val;

#pragma omp for
        for (lo i_tst = tst_elem_start; i_tst < tst_elem_end; i_tst++) {
            triangles_to_geometry_tst(i_tst, 0, 0, quadr_nodes_tst);
            for (lo i_trl = 0; i_trl < trl_spelems_count; i_trl++) {
                if (i_tst != i_trl) {
                    triangles_to_geometry_trl(i_trl, 0, 0, quadr_nodes_trl);

                    const lo &row = i_tst;
                    const lo &col = i_trl;

                    val_curr = 0;
                    val_next = 0;

                    for (lo delta = 0; delta < n_blocks; delta++) {
                        val_prev = val_curr;
                        val_curr = val_next;
                        get_local_contributions_treg_sreg(&val_next,
                            delta+1, i_tst, i_trl,
                            quadr_nodes_tst, quadr_nodes_trl);

                        matrix_val = -val_prev + 2*val_curr - val_next;

                        lo max_block = n_blocks - delta;
                        for (lo block = 0; block < max_block; block++) {
                            lo block_row = delta + block;
                            lo block_col = block;
                            sc x_val = x_perm.get(col, block_col);
                            sc y_val = alpha * matrix_val * x_val;
                            y_perm.add(row, block_row, y_val);
                        }
                    }
                }
            }
        }
    }
}
```

Listing 8: Method performing the apply operation of the fully regular component of the single layer matrix



(a) One `i_tst` loop iteration (b) One `i_trl` loop iteration (c) One `delta` loop iteration

Figure 12: Single layer matrix entries (at least partially) calculated in fully regular component application during one iteration of given loops

elements in parallel. For each of them we first map the quadrature nodes from the reference triangle to the test element using the `triangles_to_geometry_tst` method. Then we iterate through all the trial elements, avoiding the space-singular case (when test and trial elements are identical) and using the `triangles_to_geometry_trl` method we map the quadrature nodes to the trial element.

Then we loop through all the temporal element differences `delta` and calculate the value of the corresponding matrix entry. As mentioned in Section 2.3.1, we store the local contribution values from previous `delta`s and reuse them.

For calculation of the matrix entry value for a specific `delta`, we need to know the local contributions from `delta-1`, `delta`, and `delta+1`. These are stored in variables `val_prev`, `val_curr`, and `val_next`, respectively. In each iteration of the `delta` loop we shift the values and calculate only the value we do not yet know, corresponding to `delta+1`. Using the three values of the local contributions we calculate the value of the matrix entry.

Finally, using the now known value of the matrix entry, we iterate through all the blocks on the block `delta`-subdiagonal, multiply the entry with corresponding entry from the vector `x` and add the result to a corresponding entry in the vector `y`.

The matrix entries at least partially calculated during a single iteration of the `i_tst`, `i_trl` and `delta` loops are visualized in Figures 12a–12c, with the utilized entries of vectors `x` and `y` highlighted.

Other components are applied in a very similar way, utilizing different functions for calculating the heat kernel antiderivatives. The time-regular space-singular component apply considers only the identical test and trial elements. The time-singular component also has to choose an appropriate quadrature scheme depending on the relative configuration of the test and trial elements. It performs both the first and second time-singular contributions, which are located on the main block diagonal and the first block subdiagonal of the matrix.

On-the-fly application of the other main boundary element matrices is also similar. On top of what was mentioned, the component apply methods have to find the indices of the element vertices to place the local contribution values to correct rows and/or columns in the matrix, multiply it with appropriate entries from vector `x` and add the results to the appropriate entries of `y`. Moreover, in the case of matrices $\mathsf{K}_h^{\top_s}$ and D_h, each thread creates its private vector `y` to

Figure 13: Block vector permutation

avoid race conditions when trying to access the shared vector `y`. The private partial results are then added together.

5.1.4 Permuting the block vectors

Let us review again the innermost loop, which iterates through all the blocks on the block `delta`-subdiagonal, multiplies the calculated matrix entry value with the corresponding entry of vector `x` and adds the result to a corresponding position in vector `y`, as illustrated in Figure 12c. The elements of block vectors `x` and `y` used in consecutive iterations are located in completely different memory locations, resulting in poor data locality. Considering the parallelization of the `i_tst` loop, false sharing[4] can occur in vector `y` if OpenMP selects scheduling with small chunk size, possibly resulting in large performance degradation.

To increase data locality and avoid the possible false sharing, we rearrange the data in the block vectors, as depicted in Figure 13. We permute the dimensions of the block vector, transforming it from having E_t blocks each containing E_s elements to a block vector of E_s blocks with E_t elements each. Viewing the block vector as a matrix, we are performing a transposition of the matrix. The consecutive entries through which the innermost loop iterates are then stored in consecutive locations in a contiguous chunk of memory, improving data locality. Indexing of the block vectors is adjusted accordingly.

The block vector permutation is implemented using a tiled matrix transposition algorithm. Unfortunately, we could not use existing implementation of matrix transposition from Intel MKL because of the data layout in the block vector.

5.1.5 The apply method

The `apply` method on the `uniform_spacetime_be_matrix_onthefly_cpu` class first performs several checks for compatibility of the vector dimensions. Then it permutes both vectors `x` and `y` and scales them by the scalars `alpha` and `beta`, respectively. Then the `apply_cpu` method is called, which calls the three methods performing application of the three components.

5.2 GPU on-the-fly matrix

The GPU on-the-fly matrix has similar structure to the CPU version. The GPU matrix is represented by the `uniform_spacetime_be_matrix_onthefly_gpu` class, which inherits from the

[4]False sharing occurs when multiple cores write to memory locations located in the same cache line

```
uniform_spacetime_tensor_mesh spacetime_mesh( ... );
uniform_spacetime_tensor_mesh_gpu gpu_mesh(spacetime_mesh);

// ...

uniform_spacetime_be_matrix_onthefly_gpu V(kernel_v, space_p0, space_p0, gpu_mesh);

V.apply( ... );
```

Listing 9: Creation of the GPU on-the-fly matrix

`uniform_spacetime_be_matrix_onthefly_cpu` class. It is also a template class with template parameters equivalent to the CPU version.

An example usage of the class is shown in Listing 9. The user first needs to create a GPU-resident mesh, which is represented by the `uniform_spacetime_tensor_mesh_gpu` class and can be created from an instance of the `uniform_spacetime_tensor_mesh` class. The GPU mesh is an additional mandatory parameter of the on-the-fly GPU matrix constructor. We implemented four versions of the GPU algorithm, a user can specify which one to use via another optional constructor parameter. After it is constructed, the instance can be used the same way as the CPU version.

We accelerated only the application of the fully regular component, since it takes the most time to compute. The application of the other components can run on the CPU while the fully regular component is being applied on the GPU. Although the application of the fully regular contribution is accelerated, it is still the bottleneck, therefore acceleration of the other components is not necessary.

As mentioned in the previous section, we used CUDA to accelerate the algorithm. The code was implemented with multiple-GPU systems in mind. We do not use unified memory, all data movements between the host and device memory are explicit.

5.2.1 Compilation of GPU code

To make the BESTHEA library suitable even for machines without a GPU and CUDA installed, the GPU-specific classes are provided as an optional feature. To enable it, the user needs to provide the `-DUSE_CUDA` flag to the `cmake` command when compiling the library. If the flag is set, we enable the CUDA language and the nvcc compiler used to compile the `*.cu` source files, and add the GPU-specific classes to compilation.

5.2.2 GPU mesh

For performing the computations on the GPU, we need the mesh data to be present in the GPU memory. For this we created the `uniform_spacetime_tensor_mesh_gpu` class, which acts as a manager of the GPU-resident mesh data.

It hosts two other nested structures, `mesh_raw_metadata` and `mesh_raw_data`. The former contains the number of elements and nodes of the spatial mesh, the number of temporal elements and the timestep length. The latter stores pointers to the GPU-resident spatial mesh data, which are the node coordinates, mapping from elements to nodes, element normals and element areas.

The class contains one instance of the `mesh_raw_metadata` structure and a `std::vector` of `mesh_raw_data` containing one entry per GPU device.

An instance of the `uniform_spacetime_tensor_mesh_gpu` class has to be created from an instance of the `uniform_spacetime_tensor_mesh` class. The constructor populates the metadata, allocates memory for the mesh data on all available GPUs and copies the data to their memory.

5.2.3 Quadrature data structures for GPU

To store the quadrature node coordinates in the reference element and the corresponding weights in the GPU memory, we created the `quadrature_reference_raw` structure. It contains only the arrays with the quadrature node coordinates along with an array with the weights. It is a template structure with the template parameter being an `int` specifying the quadrature order. Using a `constexpr` function `qo2qs` the quadrature order is converted to quadrature size (number of nodes and weights), which is used as the size of the arrays.

Due to the expected access pattern, we store the quadrature reference on the GPUs in the constant memory. We have a GPU-constant quadrature reference variable for each of the four available quadrature orders (1, 2, 4, 5) and an associated boolean variable denoting if the quadrature reference of the corresponding order has already been initialized.

In the constructor of the `uniform_spacetime_be_matrix_onthefly_gpu` class we (among other things) check if the quadrature reference of the given order has been initialized. If it was not, we initialize it by copying the data from the `quadrature_reference` in the base class to the GPU-constant variable and update the corresponding boolean. This way we do not need to copy the quadrature reference to the GPU every time a GPU on-the-fly matrix is created. We only copy the reference nodes and weights for the regular quadrature, since the regularized quadrature is not used in the GPU-accelerated code.

Similar to the CPU version we also have a structure for storing the quadrature node coordinates mapped to the specific element, `quadrature_nodes_raw`. It contains three arrays with the x, y and z coordinates of the mapped nodes, their size is specified by the template argument the same way as in the `quadrature_reference_raw` structure.

The reason why we have separate structures for the CPU and GPU versions is, that the CPU quadrature reference utilizes `std::vector`, which is not supported in the GPU code. Also, the GPU only needs the regular quadrature scheme, which is insufficient for the CPU version.

5.2.4 Vectors in GPU memory

For the apply operation we need the block vectors `x` and `y` to be present in the GPU memory space. The memory for them is allocated only once in the lifetime of a GPU on-the-fly matrix class instance, right in the constructor. In the `apply` method we just use the once allocated memory buffers. The memory is freed in the destructor of the class.

The memory for the block vectors is allocated using the `cudaMallocPitch` function, which allocates linear memory for storage of 2D data, adding paddings to ensure proper alignment for best performance in accessing the data. The function returns a pitch (stride), which denotes the number of bytes between starts of each block of the block vector. For convenience we convert this value to the number of entries (by dividing it by the size of entry, `sizeof(sc)`) and call it the leading dimension. It is important to note, that in general we can not always convert the pitch (number of bytes) into leading dimension (number of entries), mainly when the entry size does not divide the alignment (usually 512 bytes) without a remainder.

For managing the vectors in the GPU memory we created the `gpu_apply_vectors_data` structure, which contains the pointers, corresponding pitches and leading dimensions of both block vectors `x` and `y` for all available GPU devices. It also stores copies of the vectors in one contiguous chunk of the host memory.

5.2.5 The apply method, GPU algorithm versions

In the `uniform_spacetime_be_matrix_onthefly_gpu` class there are methods for performing the GPU-accelerated on-the-fly matrix-vector multiplication only for the fully regular component. Application of the other components is passed on to the CPU on-the-fly matrix.

We developed four versions of the GPU algorithm for applying the fully regular component, each with different thread logic and data movement strategies. In the following text we explain their functionality and in the Section 6.8 we will compare their performance. The used version of the GPU algorithm can be specified by a parameter of the class constructor.

For each of the main boundary element matrices and each of the four algorithm versions there is a kernel function (marked `__global__`) implementing the GPU on-the-fly apply algorithm, resulting in a total of 16 of those functions. The algorithm version is specified by the function name, the matrix by parameters of the function, utilizing function overloading. These functions are templated with a template parameter denoting the used quadrature order. Their parameters are the three parameters specifying the main boundary element matrix (heat kernel, test and trial spaces), the block vectors `x` and `y` along with their leading dimension, the scalar `alpha`, the starting test element, mesh data and metadata and the heat kernel parameters (containing mainly the heat capacity constant).

In the explanations we will again focus mostly on the single layer matrix V_h and then mention the differences in other matrices. We explain the main principles of the implementations and do not discuss edge cases. We also for simplicity assume only one GPU device in the system. Here we provide the source code only for the first version of the GPU algorithm, the remaining three versions can be found in Appendix B.

Since the implementation of the functions calculating the fully regular local contributions on GPU is very similar to the CPU version, they will not be described in the following text.

GPU algorithm version 1

The first GPU algorithm version (shown in Listing 10) is the most similar to the CPU version. We launch the kernel as a one-dimensional grid containing as many threadblocks as there are test elements, with each of them being assigned exactly one test element based on the value of `blockIdx.x`. Similar to how in the CPU algorithm the `i_tst` loop iterates through test elements, computation originally handled by one iteration of the loop is here handled by one threadblock. It is important to think about the threadblock as a whole, not to think about each thread separately.

Each threadblock allocates a `__shared__` variable of type `quadrature_nodes_raw` for storing the quadrature node coordinates mapped to the test element. All threads in a threadblock then perform the mapping cooperatively.

The threadblock then loops through all the trial elements, shifting itself by `blockDim.x` elements in each iteration. There each thread in the threadblock gets assigned one of the trial elements and maps the reference quadrature nodes to the this trial element. Then the threads loop through all the `delta`s, in each iteration calculating the matrix value corresponding to

```
template< int quadr_order >
__global__ void g_apply_gpu_treg_sreg_ver1(
    const besthea::bem::spacetime_heat_sl_kernel_antiderivative * _hka,
    const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 > * _tst_space,
    const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 > * _trl_space,
    const sc * x, lo ld_x, sc * y_perm, lo ld_y_perm, sc alpha, lo i_tst_begin,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_metadata mesh_metadata,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_data mesh_data,
    const ns_gpu_helpers::heat_kernel_parameters kp) {

    constexpr int tpbx = tpb_V[1][quadr_order].x;

    __shared__ ns_gpu_helpers::quadrature_nodes_raw<quadr_order> shmem_quadr_nodes_tst;
    __shared__ volatile sc shmem_y_vals[tpbx];

    const lo &n_blocks = mesh_metadata.n_temporal_elements;
    const lo &n_elems = mesh_metadata.n_elems;
    const lo i_tst = i_tst_begin + blockIdx.x;
    const unsigned int &tid = threadIdx.x;

    shmem_y_vals[tid] = 0;
    d_triangles_to_geometry_000_tst_shmem(i_tst, mesh_data, shmem_quadr_nodes_tst);
    __syncthreads();

    sc matrix_val;
    sc val_prev;
    sc val_curr;
    sc val_next;

    ns_gpu_helpers::quadrature_nodes_raw<quadr_order> quadr_nodes_trl;

    for (lo i_trl = threadIdx.x; i_trl < n_elems; i_trl += blockDim.x) {
        d_triangles_to_geometry_000_trl(i_trl, mesh_data, quadr_nodes_trl);

        const lo &row = i_tst;
        const lo &col = i_trl;

        val_curr = 0;
        val_next = 0;

        for (lo delta = 0; delta < n_blocks; delta++) {
            val_prev = val_curr;
            val_curr = val_next;
            d_get_local_contributions_treg_sreg_sl_p0_p0(&val_next, delta+1, i_tst, i_trl,
                shmem_quadr_nodes_tst, quadr_nodes_trl, mesh_metadata, mesh_data, kp);

            matrix_val = ((i_tst == i_trl) ? (0) : (-val_prev + 2*val_curr - val_next));

            lo max_block = n_blocks - delta;
            for (lo block = 0; block < max_block; block++) {
                lo block_row = delta + block;
                lo block_col = block;
                shmem_y_vals[tid] = matrix_val * x[block_col * ld_x + col];
                __syncthreads();
                d_reduce_sum<tpbx>(shmem_y_vals);
                if(tid == 0)
                    y_perm[block_row + ld_y_perm * row] += alpha * shmem_y_vals[0];
            }
        }

        shmem_y_vals[tid] = 0;
        __syncthreads();
    }
}
```

Listing 10: GPU on-the-fly algorithm version 1

```
template<int tpbx>
__device__ void d_reduce_sum(volatile sc * shmem_vals) {

    int curr_thread_count = tpbx / 2;
    int tid = threadIdx.x;

    while(curr_thread_count > 32) {
        if(tid < curr_thread_count) {
            shmem_vals[tid] += shmem_vals[tid ^ curr_thread_count];
        }
        __syncthreads();
        curr_thread_count /= 2;
    }

    if(tid < 32) {
        if(tpbx >= 64) shmem_vals[tid] += shmem_vals[tid ^ 32];
        if(tpbx >= 32) shmem_vals[tid] += shmem_vals[tid ^ 16];
        if(tpbx >= 16) shmem_vals[tid] += shmem_vals[tid ^  8];
        if(tpbx >=  8) shmem_vals[tid] += shmem_vals[tid ^  4];
        if(tpbx >=  4) shmem_vals[tid] += shmem_vals[tid ^  2];
        if(tpbx >=  2) shmem_vals[tid] += shmem_vals[tid ^  1];
    }

    __syncthreads();
}
```

Listing 11: Parallel reduction within a threadblock on GPU

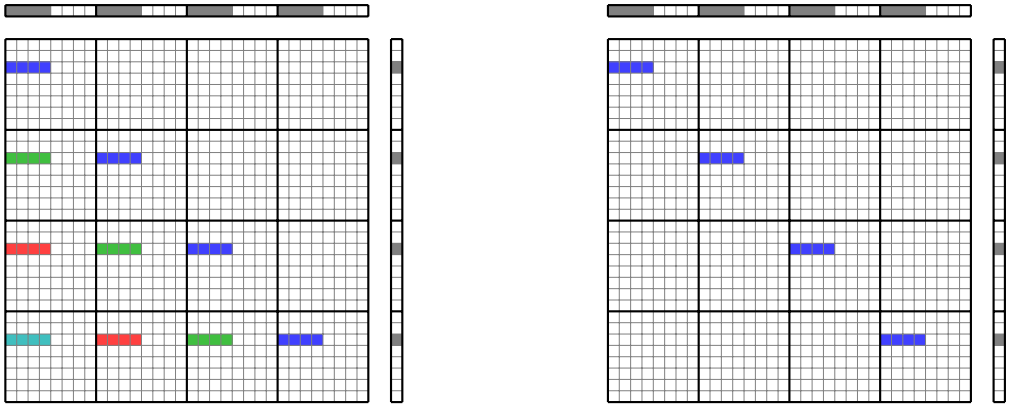
the test and trial elements and the `delta`, again reusing the once calculated local contribution values. If the test and trial elements happen to be identical, we set the matrix value to 0, because the time-regular space-singular component is computed on the CPU.

For each `delta` the threads loop through all the blocks on the block `delta`-subdiagonal. For each of them they read the values from the vector `x` and multiply them with the matrix values. The results then need to be added to a single entry in the vector `y`. We cannot just naively perform the addition, because race conditions would appear. Therefore we perform a parallel reduction utilizing shared memory (shown in Listing 11). After the reduction is completed, the first thread in the threadblock adds the result to the entry of vector `y`.

We have tried several ways of adding the results to the entry of vector `y`, which includes performing the additions atomically, performing a reduction only within a warp and then adding the results atomically, reducing the warp-reduction results again using a warp-reduction, but the classic parallel reduction turned out to be the most performant.

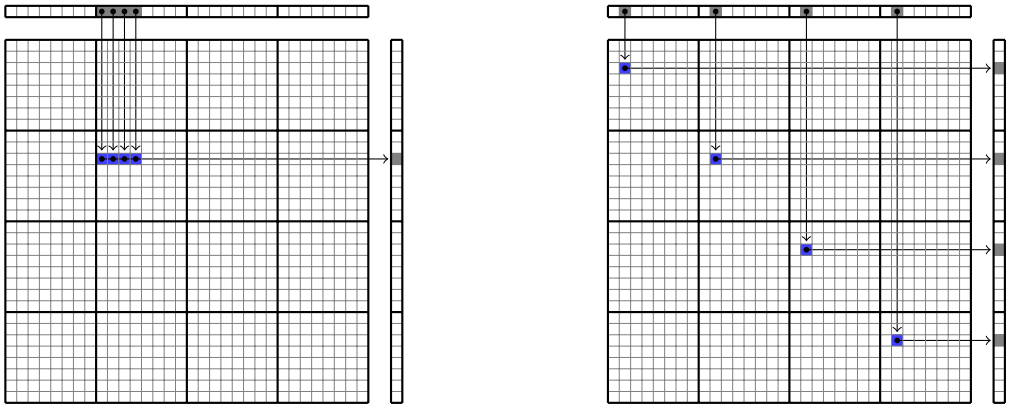
Matrix entries calculated by all threads within a threadblock in one iteration of the `i_trl` loop are highlighted in Figure 14a, the entries calculated in an iteration of the `delta` loop are highlighted in Figure 14b. The data movement in one iteration of the `block` loop is visualized in Figure 15a. In the figures we include the identical test and trial elements, since we perform everything equally as for other element pairs, with the only difference of setting the matrix value to zero.

As in the CPU implementation, we explore the possibilities of permuting the block vectors `x` and `y`. Because neighboring threads access neighboring entries in the block vector `x`, it should not be permuted. Through `y` we iterate by blocks, permuting it should therefore be beneficial.



(a) One `i_trl` loop iteration (b) One `delta` loop iteration

Figure 14: Matrix entries calculated by all threads within a threadblock during one iteration of given loops in GPU algorithm versions 1 and 2



(a) GPU algorithm version 1 (b) GPU algorithm version 2

Figure 15: Data movement when contributing the matrix entries to the vector `y` in GPU algorithm versions 1 and 2

GPU algorithm version 2

Up until the calculation of matrix entries, everything happens the same way in the second GPU algorithm as in the first one. The multiplication with vector `x` and addition to `y` however works differently. The large number of synchronization points (calls to `__syncthreads()`) required in the reductions might decrease the performance, therefore we try to avoid it.

When each thread calculates its corresponding matrix entry value, it stores it into the shared memory to make it accessible for all threads in a threadblock. After a synchronization to make sure all threads have computed their entries, we start contributing the matrix entries to the vector `y`.

Each thread is assigned a different block on the `delta`-subdiagonal. The threadblock loops through all the calculated matrix entries in the shared memory, for each of them reads a corresponding entry from the vector `x`, multiplies it with the matrix entry and adds the result to a local private variable tracking the total contribution to the corresponding entry in the vector `y`.

After the loop iterating through the calculated matrix entries is finished, the totals are added to the vector `y`. The data movement in one iteration of the matrix entry loop is visualized in Figure 15b.

We have thus successfully reduced the required number of synchronization points. However, the main drawback of this algorithm is, that for larger values of `delta`, when there are only a few blocks on the `delta`-subdiagonal, many threads are not utilized and are idle. The same happens when the block dimensions[5] of the matrix are small.

Looking at the data access pattern in the block vectors `x` and `y`, we should permute both of them. Permutation of `y` is expected to be less noticeable, since it is accessed a lot less frequently than `x`.

GPU algorithm version 3

The third version of the GPU algorithm is similar to the first one. The main difference is, that we launch the kernel as a one-dimensional grid of two-dimensional threadblocks, and assign each threadblock a range of test elements, as opposed to only one in the first version. The number of the test elements handled by each threadblock is defined by the threadblock size in the x dimension (`blockDim.x`).

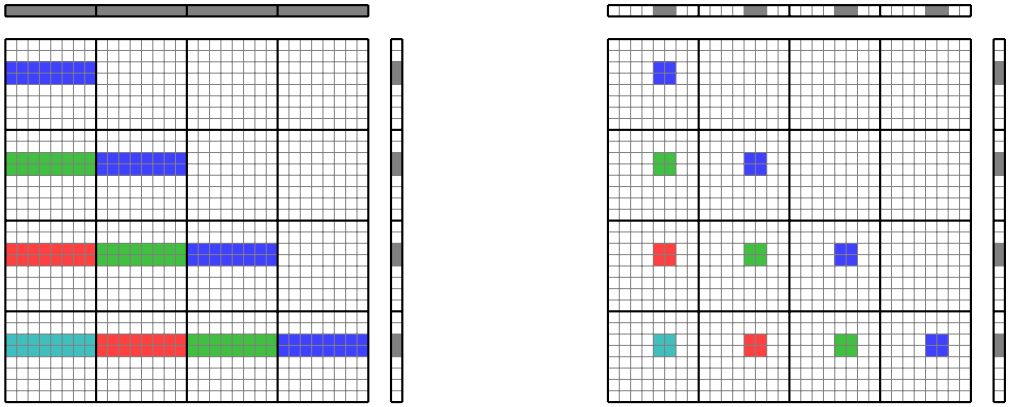
The main idea is, that in the first algorithm, we perform a total of E_s^2 mappings of the quadrature nodes to the trial elements – we have to do the mappings in each threadblock (i.e. for each test element) separately, as data cannot be easily shared between threadblocks. By dealing with several test elements in a threadblock at once, the number of performed mappings to the trial elements drops by a factor `blockDim.x`. The number of mappings to the test elements does not change.

Each threadblock is assigned a strip of matrix entries to calculate and contribute, as visualized in Figure 16a. We then loop through all trial elements, shifting the rectangle of test and trial elements currently being dealt with by `blockDim.y`. One such rectangle (in this case a square) is visualized in Figure 16b.

At the start of the kernel we map the reference quadrature node coordinates to all corresponding test elements and store them in shared memory. In each iteration of the `i_trl` loop we map the reference quadrature to the trial elements and also store it in shared memory. Then we do the `delta` loop, and in each iteration calculate the values of corresponding matrix entries. Inside it there is the `block` loop iterating through all the blocks on the `delta`-subdiagonal. In each iteration we read the values from block vector `x`, multiply them with the matrix values, perform a reduction in each row and add the results to the vector `y`, as visualized in Figure 17a.

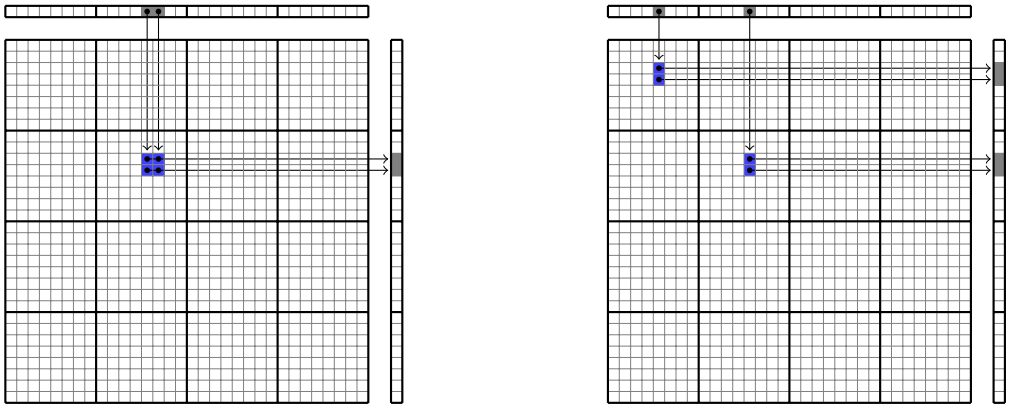
As there can be currently up to 49 nodes and weights in the regular quadrature scheme, storing the mapped node coordinates can take up a large portion of shared memory. For larger threadblocks the shared memory does not even have sufficient capacity to hold all the requested data and the kernel launch fails. The higher number of test elements per threadblock also means, that the number of threadblocks will be smaller, decreasing the granularity, lowering occupancy and possibly not fully utilizing the GPU.

[5] As mentioned before, by block dimensions we mean the number of block rows and columns in the matrix



(a) Whole kernel (b) One `i_trl` loop iteration

Figure 16: Matrix entries calculated by all threads within a threadblock in given parts of the GPU algorithm versions 3 and 4



(a) GPU algorithm version 3 (b) GPU algorithm version 4

Figure 17: Data movement when contributing the matrix entries to the vector `y` in GPU algorithm versions 3 and 4

GPU algorithm version 4

The fourth version of the GPU algorithm can be thought of as an improvement of the second or third version, combining improvements made in both of them. It works the same way as the third version up until the computation of matrix entries. Then it behaves differently, similar to how version 2 differs from version 1, replacing parallel reduction with a loop and dealing with multiple blocks at the same time.

In each iteration of the `delta` loop we calculate the matrix entries as in the third version, but we store them into shared memory to make them accessible by all threads in the threadblock. Each thread is then assigned a test element (row in the matrix block) and a block on the `delta`-subdiagonal. The test element is specified by `threadIdx.x`, while the block by `threadIdx.y`. Then we loop through all the trial elements for which we have calculated the matrix entries. For each of them we read a corresponding entry from the vector `x`, multiply it with the value of the matrix entry and accumulate it to a local variable. After the trial loop is finished, we contribute

the results into a corresponding entry in vector `y`. Data movement during one iteration of the trial loop is visualized in Figure 17b.

This approach reduces the drawback of the second version, but the problems with limited shared memory space and granularity persist.

Other matrices

The GPU code for applying other matrices is mostly similar to the single layer matrix. We need to find the vertex indices of the test and/or trial elements and calculate multiple local contribution values (partial values of matrix entries) at once during the numerical quadrature. For matrices K_h and D_h we need to read three entries from the vector `x`. For $\mathsf{K}_h^{\top_s}$ and D_h we add the three contributions to vector `y` atomically to avoid race conditions.

The apply method

The `apply` method of the `uniform_spacetime_be_matrix_onthefly_gpu` class first permutes and scales the vectors `x` and `y`. Then we copy the data from the block vector `x` to a single contiguous chunk of host memory, which we copy to the device memory, and fill the device vector `y` with zeros. Based on the chosen algorithm version and quadrature order we launch a corresponding kernel function to compute the application of the fully regular component. Then the vector `y` is copied from the device to a contiguous chunk of host memory. Copying between host and device and the kernel launch are asynchronous operations, meaning we only submitted them to the GPU, but the actual execution might happed at a later time.

After we submit all the work to the GPU, we call the CPU functions to perform the application of the time-regular space-singular and time-singular components. This is being computed at the same time as the fully regular component is being executed on the device. After the CPU work finishes, we wait for the GPU to finish its computations as well, and then add the vectors `y` from the CPU and GPU together to get the final result.

5.2.6 Multiple GPUs, CPU-GPU load balancing

The developed code is able to utilize multiple GPUs in a system. We split all the test elements into as many approximately equally-sized chunks as there are GPU devices. To the kernel functions we provide a parameter specifying the start of the chunk computed by the given GPU, the size of the chunk is specified by the grid dimension. We assume all GPUs in the system have equal performance.

The application of the fully regular component on the GPU is still the bottleneck and the CPU is idle while waiting for the GPU to finish. We therefore implemented a CPU-GPU load-balancing, which splits the application of the fully regular component between the CPU and the GPUs in such a way that the CPU is idle for the least amount of time. This decreases the amount of work the GPUs have to do, thus reducing the total time of the computation.

For this purpose the `apply_load_distribution` class was created, which handles the CPU-GPU load balancing. Based on the time of computation on the CPU and GPUs it calculates the optimal distribution of the test elements among the devices.

Both the CPU and GPU have a portion of time which stays constant, independent of the number of assigned test elements. For CPU this is the application of the time-regular space-singular and time-singular components, for GPU it is the time it takes to copy the vectors to

and from the device. We assume that the time it takes to compute the application of the fully regular part scales linearly with the number of assigned test elements.

We set $T_{C,1}$ and $T_{G,1}$ to the measured average time it takes to perform the application of the fully regular component for one test element on the CPU and GPU, respectively. We further set $T_{C,c}$ and $T_{G,c}$ to the measured times of the constant portions of the CPU and GPU code, respectively. We denote N_C and N_G the number of test elements assigned to the CPU and GPU, respectively, with the constraint $N_C + N_G = E_s$. To find the optimal load distribution, we need to find a solution of the equation

$$T_{C,c} + N_C T_{C,1} = T_{G,c} + N_G T_{G,1}$$

with respect to N_C. This equation says that the time spent by all the CPU computations should be equal to the time spent by the GPU work. By rearranging the equation we get

$$N_C = \frac{E_s T_{G,1} + T_{G,c} - T_{C,c}}{T_{G,1} + T_{C,1}}.$$

Rounding the solution down to a nearest integer, handling all possible edge cases and constraints for the number of test elements assigned to the GPU, we update the number of test elements assigned to the CPU.

In the first call of the `apply` method we set the number of test elements assigned to the CPU to the result of function `omp_get_max_threads()`, which returns the number of threads utilized by an OpenMP parallel region. Within the `apply` method we measure the time it takes to compute all the different sections. At the end of the method we update the load distribution using the measured times. The next invocation will use the updated number of test elements assigned to the CPU. The GPUs are assigned all the remaining test elements. The load distribution is updated every time the `apply` method is called.

The number of test elements handled by the CPU version is set by the previously mentioned parameter of the fully regular apply method. The CPU-GPU load balancing pays off especially when calling the `apply` method several times in a row, for example in an iterative solver. We analyze the effect of the load-balancing on the execution time in Section 6.9.

6 Numerical experiments

In this section we conduct several numerical experiments to test performance of our CPU and GPU implementations in various environments, to compare several implementation approaches, and to test the accuracy of the solution.

Time is measured using `std::chrono::steady_clock` for CPU workloads and `cuda_events` for GPU-related tasks. For most experiments the elapsed time is computed as an average of 10 runs of the monitored section with 2 preceding runs not included in the timing due to the possibility of additional overhead.

We are using a spatial mesh representing a cube centered at the origin with side length of 2, time interval $(0, 1)$ and the heat capacity constant $\alpha = 1$, unless stated otherwise. We refine the space-time mesh to get results on multiple problem sizes while fixing the ratio h_x^2/h_t, which guarantees optimal convergence rate for solving the system of equations arising from the space-time boundary element method for the heat equation [14, p. 23]. We use two base spatial meshes consisting of 12 and 24 elements each (2 and 4 triangles per side of the cube, respectively), which we refine to get the mesh with the desired number of elements. If not specified, for the spatial integrals we use numerical quadrature with order 4. At all times we use double precision representation of floating point numbers, i.e. we set `sc=double`. All the experiment results and bash scripts used to run the experiments are available in the attachment (see Appendix A).

6.1 Machines

Some of the experiments were run on multiple machines to compare the performance on various types of CPUs and GPUs.

The machine we conduct most of the experiments on is a GPU accelerated node of the Barbora cluster at IT4Innovations National Supercomputing Center in Ostrava. The GPU node has two Intel Skylake Gold 6126 12-core CPUs clocked at 2.6 GHz, a total of 192 GB of DDR4 physical memory and is equipped with 4 GPU accelerators NVIDIA Tesla V100-SXM2. In the following text we will refer to this machine as the Barbora GPU node.

For some CPU-only workloads we use a regular computational node of the the Barbora cluster. This node has two Intel Cascade Lake 6240 18-core processors clocked at 2.6 GHz and 192 GB of DDR4 physical memory. This machine will be referred to as the Barbora CPU node. More information about IT4Innovations infrastructure can be found in the IT4Innovations documentation [7].

The final machine is a representative of a higher performance laptop. It is equipped with an 8-core AMD Ryzen 7 4800H CPU, which we run at stable 2.9 GHz, and an NVIDIA GeForce GTX 1650 Ti GPU. Windows 10 is installed on this machine, but we run all the experiments in Ubuntu 18.04 inside a WSL 2 environment (Windows Subsystem for Linux), which is similar to a virtual machine. We will refer to this machine simply as the laptop.

6.2 Compilation

On Barbora GPU and CPU nodes, `C++` sources are compiled with Intel compiler 19.1.3.304, CUDA sources are compiled with nvcc (CUDA 11) with host compiler set to the default – g++, which is available in version 10.2.0. Due to incompatibility of the combination of nvcc, Intel compiler and Eigen library, the CUDA host compiler could not be set to the Intel compiler[6].

[6] See the issue `https://gitlab.com/libeigen/eigen/-/issues/2180`

Additional Intel vectorization flags `-xcore-avx512` and `-qopt-zmm-usage=high` are used on Barbora nodes.

On the laptop we compile the `C++` code with g++ 9.3.0, CUDA sources are compiled with nvcc (CUDA 11) with the default host compiler g++ 9.3.0.

Optimization flag `-O2` is used at all times. For more details about compilation see the CMake files in the attachment.

6.3 Permutation of block vector

As we mentioned in Section 5, for the purpose of the on-the-fly matrix-vector multiplication we sometimes permute the block vectors `x` and `y`, which should improve data locality and prevent possible false sharing. In this experiment we compare the performance of the algorithms for all four combinations of vector permutations for both the CPU and GPU implementations.

We used quadrature order of 1 to suppress the time of numerical integration and highlight the differences in memory access strategies. We only measure the time of the actual on-the-fly matrix-vector multiplication and ignore the time it takes to permute the vectors and additional overheads such as memory transfers to/from GPU.

The CPU implementation was run on the Barbora CPU node using a mesh with 128 temporal and 3072 spatial elements. The variants of the GPU implementation were run on the Barbora GPU node utilizing all four GPUs using mesh consisting of 256 temporal and 6144 spatial elements. For the GPU version we only measure the time of the regular component, since this is the only one computed on the GPU. We also make sure the CPU-GPU load balancing is turned off, so that only the GPUs are used for calculation of the regular part. We use threadblock dimensions of 128 for GPU versions 1 and 2 and 16×8 for versions 3 and 4. The measured times (in seconds) are shown in Tables 4a and 4b for the CPU versions with 18 and 36 threads, respectively, and in Tables 4c–4f for the four GPU versions.

For the 18-thread CPU version we can see that permuting either vector `x` or `y` comes with performance benefit, permuting both is the optimal choice, which was expected. For the 36-thread CPU version the same statement holds with the exception of matrices V_h and K_h with only the vector `x` permuted, when the performance dropped significantly. This was not a single fluctuation and was consistently measured on multiple computational nodes. The reason to this is expected to be NUMA (non-uniform memory access) effect.

The first GPU algorithm is negatively affected by permuting the vector `x`, while permuting `y` has small benefit, mainly on matrix V_h. GPU algorithm number 2 largely benefits from permuting vector `x`. With the vector `x` permuted, permuting the vector `y` has different effects on different matrices – about 25 % longer runtime with the matrix V_h, but about 1-6 % shorter for the other matrices. We choose not to permute `y` because of the more significant performance difference on the matrix V_h. The third version of GPU algorithm benefits from permuting `x` and with the negligible exception in matrix K_h it is not beneficial to permute `y`. Version 4 of the GPU implementation is negatively affected by permuting the vector `y`, permuting `x` leads to a small improvement of computation time.

The optimal choices of permuting the vectors are summed up in Table 5 and will be used in all further experiments. This experiment was also run on the laptop with very similar results. The result for the GPU kernel version 2 is different from the expectation, which was that it would be beneficial to permute both vectors. The difference in measured times is not large, but is significant enough to make the conclusion.

Table 4: Performance comparison of permuting vectors in on-the-fly matrix-vector multiplication (computation time in seconds)

(a) CPU, 18 threads

Permuted		Matrix			
x	y	V_h	K_h	$K_h^{\top_s}$	D_h
no	no	12.98	21.25	23.51	49.11
no	yes	10.06	15.66	18.12	42.02
yes	no	11.14	14.62	19.73	45.86
yes	yes	9.38	13.52	16.33	38.67

(b) CPU, 36 threads

Permuted		Matrix			
x	y	V_h	K_h	$K_h^{\top_s}$	D_h
no	no	6.46	10.60	12.00	25.02
no	yes	5.04	8.21	9.20	21.54
yes	no	32.36	20.86	10.01	24.20
yes	yes	4.72	6.86	8.33	19.71

(c) GPU, algorithm version 1

Permuted		Matrix			
x	y	V_h	K_h	$K_h^{\top_s}$	D_h
no	no	3.36	5.81	7.26	15.52
no	yes	2.98	5.78	7.26	15.55
yes	no	5.49	8.05	8.69	17.53
yes	yes	5.41	8.39	8.71	17.56

(d) GPU, algorithm version 2

Permuted		Matrix			
x	y	V_h	K_h	$K_h^{\top_s}$	D_h
no	no	3.20	9.42	3.19	10.41
no	yes	3.10	9.35	3.12	10.36
yes	no	1.24	1.94	2.06	3.80
yes	yes	1.56	1.89	1.93	3.77

(e) GPU, algorithm version 3

Permuted		Matrix			
x	y	V_h	K_h	$K_h^{\top_s}$	D_h
no	no	9.53	15.41	17.81	25.97
no	yes	9.53	14.42	18.17	26.68
yes	no	9.29	11.17	17.70	23.74
yes	yes	9.44	11.09	17.90	24.36

(f) GPU, algorithm version 4

Permuted		Matrix			
x	y	V_h	K_h	$K_h^{\top_s}$	D_h
no	no	1.99	4.94	3.82	7.21
no	yes	2.53	5.75	3.93	7.55
yes	no	1.95	4.29	3.75	7.24
yes	yes	2.61	4.71	3.82	7.39

Table 5: Optimal choices of vector permutations for on-the-fly matrix-vector multiplication

implementation	permute x	permute y
CPU	yes	yes
GPU version 1	no	yes
GPU version 2	yes	no
GPU version 3	yes	no
GPU version 4	yes	no

Table 6: Comparison of time it takes to apply the fully regular (FR), time-regular space-singular (TRSS) and time-singular (TS) components of the single layer matrix

		Elapsed time [s]			Ratio [%]			Factor [-]		
E_t	E_s	FR	TRSS	TS	FR	TRSS	TS	FR	TRSS	TS
4	96	0.0183	0.0001	0.0057	58.70	0.78	40.52	-	-	-
8	192	0.0647	0.0004	0.0171	78.67	0.50	20.83	7.81	3.75	2.99
16	384	0.5264	0.0016	0.0579	89.84	0.28	9.88	8.14	3.97	3.38
32	768	4.3350	0.0070	0.2208	95.01	0.15	4.84	8.24	4.27	3.81
64	1536	37.6030	0.0296	1.0162	97.29	0.08	2.63	8.67	4.23	4.60
128	3072	333.4700	0.1234	4.7978	98.55	0.04	1.41	8.87	4.17	4.72

6.4 Time comparison of applying individual components

The application of the fully regular component asymptotically takes the most time, having $\mathcal{O}(E_t^2 E_s^2)$ time complexity. The time-regular space-singular and time-singular components have lower complexities of $\mathcal{O}(E_t^2 E_s)$ and $\mathcal{O}(E_t E_s^2)$, respectively. With each doubling of the number of both temporal and spatial elements, we expect the computational time to increase by scaling factors 16, 8 and 8 for the fully regular, time-regular space-singular and time-singular components, respectively.

We measured the computation times of applying the individual components and compared their ratio for several mesh refinements. We also calculated the factors by which the computation time increased with each refinement of the space-time mesh. The results for the single layer matrix are shown in Table 6, the results for other matrices were very similar.

The experiment was run using the CPU on-the-fly matrix on the Barbora CPU node. We utilized only one thread to suppress possible parallel scaling differences between the individual components. For the finest discretization we run the method only once to save computational resources.

As we expected, with finer mesh discretizations the fully regular component dominates. However, the computation time does not scale with the problem size as expected. This is probably caused by the problem sizes not being large enough. The computation time is still dominated by the calculation of matrix entries and the looping through the blocks has only little effect on the computation time. This is supported by the increasing trend of the factors. We also ran this experiment for quadrature order 1, for which the time of matrix entry computation is shorter, and the factors tend towards the expected values faster.

6.5 Parallel scaling of CPU on-the-fly apply method

The CPU on-the-fly matrix-vector multiplication algorithm is parallelized using OpenMP. In this experiment we analyze the strong scaling of the algorithm – measure the time of the `apply` method for different numbers of threads, calculate corresponding speedup and efficiency. The results are shown in Table 7.

We ran the experiment on the Barbora CPU node using the CPU implementation of the on-the-fly matrix. We used a mesh consisting of 128 temporal and 3072 spatial elements. To save resources and time, the experiment was run only once for up to 4 threads, for up to 18

Table 7: Strong parallel scaling of the CPU on-the-fly apply method

Thread count	Elapsed time [s]				Speedup [-]				Efficiency [%]			
	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h
1	337.6	444.7	498.2	965.9	1.0	1.0	1.0	1.0	100.0	100.0	100.0	100.0
2	168.8	221.8	249.2	489.8	1.9	2.0	1.9	1.9	99.9	100.2	99.9	98.5
4	84.4	109.5	124.5	245.6	3.9	4.0	4.0	3.9	99.9	101.4	100.0	98.3
8	44.4	55.5	62.5	122.9	7.5	8.0	7.9	7.8	94.9	100.0	99.5	98.1
12	28.1	37.1	41.6	82.3	11.9	11.9	11.9	11.7	99.8	99.8	99.6	97.7
18	19.2	24.7	28.8	54.8	17.5	17.9	17.2	17.6	97.6	99.8	96.1	97.8
24	14.7	18.5	21.2	41.2	22.9	23.9	23.4	23.4	95.6	99.9	97.7	97.6
36	9.8	12.5	14.1	27.7	34.3	35.5	35.2	34.7	95.2	98.7	98.0	96.6

threads it was run three times with one preceding warp-up run. We measure the time it takes to execute the whole `apply` method, including the vector permutations and scaling.

Looking at the table we can conclude, that the strong parallel scalability of the matrix application is near ideal, achieving no less than 95 % efficiency.

We also analyzed the scalability of applying the individual components and observed, that the fully regular component also scales almost ideally, which holds for the time-singular component as well. The time-regular space-singular component scales worse, having efficiency as low as 19 % for matrix $\mathsf{K}_h^{\top_s}$. This is, however, not a concern, since the application of this component takes only a fraction of the total execution time.

6.6 Optimal threadblock dimensions

The GPU kernel function needs to be configured for execution, that is the grid and threadblock dimensions need to be specified. We always choose the grid dimensions based on the number of elements in the spatial mesh, therefore there is no optimization to be made. However, the threadblock dimensions can be specified independently of the problem size.

Performance of a kernel usually largely depends on the chosen threadblock dimensions, its optimal value can vary between different kernels of different complexity and logic, devices, and many other factors. The optimal threadblock dimensions are hard to predict theoretically, we will therefore find the optimum experimentally.

We run the experiment on the Barbora GPU node and use a mesh with 256 temporal and 6144 spatial elements. We again measure only the time it takes to compute the regular part on the GPU and make sure the CPU-GPU load balancing is turned off. We searched for the optimum threadblock dimensions for each kernel version, matrix and quadrature order (as it largely affects the amount of utilized shared memory). We do not list all the measured times as the tables would be very extensive, the found optima are listed in Tables 8a–8d.

For kernel versions 1 and 2 the most common optimal threadblock size was 128 with less commonly occurring 64 and 256. There were no choices for the threadblock size that would decrease performance drastically, the worst of them took about twice the time to compute compared to the optimum.

The most common optimal threadblock dimensions for kernels 3 and 4 are 8×16. In the cases where this choice was not the optimum, it was never significantly slower, taking at most

Table 8: Measured optimal threadblock dimensions

(a) GPU kernel version 1

Quadrature order	Matrix V_h	K_h	$K_h^{\top_s}$	D_h
4	256	128	128	64
5	256	128	128	64

(b) GPU kernel version 2

Quadrature order	Matrix V_h	K_h	$K_h^{\top_s}$	D_h
4	128	128	128	64
5	128	128	128	64

(c) GPU kernel version 3

Quadr. order	Matrix V_h	K_h	$K_h^{\top_s}$	D_h
4	8×16	8×16	8×16	8×8
5	8×16	8×16	8×16	4×8

(d) GPU kernel version 4

Quadr. order	Matrix V_h	K_h	$K_h^{\top_s}$	D_h
4	8×16	4×16	4×16	2×16
5	8×16	8×16	8×16	4×16

15 % more time to finish relative to the optimal choice. There were however many choices which degraded performance drastically, some being up to 40 times slower than the optimum. Many runs with high thread count failed with an error, probably resulting from too much shared memory and registers being requested. Due to the extent of the search space we only repeated the calculation three times with one preceding warm-up run.

We also conducted the experiment on the laptop. The optimal threadblock dimensions were, as expected, mostly different. However, using optima from the Barbora GPU node for the laptop, the execution time increased by at most 5 % compared to the optimum.

6.7 Scaling on multiple GPUs

The GPU version of the matrix-vector multiplication algorithm is capable of utilizing multiple GPUs in a system. The scaling of the execution time based on the number of utilized GPUs is for the four algorithm versions and four matrices shown in Table 9.

We ran this experiment on the Barbora GPU node using a space-time mesh with 256 temporal and 6144 spatial elements. The CPU-GPU load balancing was turned off for this experiment and we measured only the computation time of the fully regular component on the GPU. The number of utilized GPUs was controlled by the environment variable `CUDA_VISIBLE_DEVICES`.

Analyzing the measured computation times and speedups, we find that the first two algorithm versions can utilize multiple GPUs very well, with ideal linear or even superlinear scalability. The third version scales almost ideally for up to 3 GPUs, adding the fourth GPU brings only a little performance benefit. The fourth version scales differently on different matrices, but on four GPUs the efficiency is not less than 85 %.

6.8 Comparison of GPU algorithm versions

In Section 5.2.5 we presented four versions of the GPU matrix-vector multiplication algorithm. Each of them has its advantages and drawbacks. Let us now experimentally find the best performing version. The computation times (in seconds) using the four algorithm versions for

Table 9: Scaling of the GPU algorithm on multiple GPUs

Algorithm version	Number of GPUs	Computation time [s]				Speedup [-]			
		V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h
1	1	21.70	31.81	42.29	98.74	1.00	1.00	1.00	1.00
	2	10.78	15.66	21.33	49.55	2.01	2.03	1.98	1.99
	3	7.29	11.33	15.44	34.59	2.97	2.80	2.73	2.85
	4	5.31	7.83	10.79	24.81	4.08	4.06	3.91	3.98
2	1	15.93	24.11	20.15	38.60	1.00	1.00	1.00	1.00
	2	7.54	11.88	9.53	19.34	2.11	2.03	2.11	1.99
	3	4.93	8.09	6.48	13.34	3.22	2.98	3.11	2.89
	4	3.68	5.83	4.67	9.69	4.32	4.13	4.31	3.98
3	1	27.90	33.56	43.09	78.58	1.00	1.00	1.00	1.00
	2	18.59	22.35	28.79	52.28	1.50	1.50	1.49	1.50
	3	9.46	11.33	14.52	26.37	2.94	2.96	2.96	2.97
	4	8.90	10.47	13.26	25.99	3.13	3.20	3.25	3.02
4	1	15.39	23.73	20.69	44.18	1.00	1.00	1.00	1.00
	2	10.24	12.13	10.59	23.31	1.50	1.95	1.95	1.89
	3	5.14	9.91	8.60	16.15	2.99	2.39	2.40	2.73
	4	4.35	6.01	5.19	11.97	3.53	3.94	3.98	3.69

several mesh refinements and the four main boundary element matrices are shown in Tables 10a–10d.

This experiment was performed on the Barbora GPU node utilizing all four GPUs with CPU-GPU load balancing turned off. We measured only the application of the fully regular component on the GPU.

From the tables we can conclude, that the versions 2 and 4 were the most performant, with the second version being the most common best choice. For matrices V_h and $\mathsf{K}_h^{\top_s}$ the fourth version starts to dominate for the finest mesh. However, the difference is less than 10 %, and the fourth version performs significantly worse for coarser meshes. Therefore we conclude the best GPU algorithm to be the version 2 and we will use it for all subsequent experiments.

We also performed the experiment on the laptop, with the same observation of versions 2 and 4 performing the best. The difference to versions 1 and 3 was, however, much less noticeable, peaking at around 20 %.

6.9 CPU-GPU load balancing

As we mentioned in Section 5.2.6, we implemented a CPU-GPU load balancing to even further reduce the time it takes to execute the `apply` method on the GPU. We will now analyze how much performance impact this technique has and observe how the computation time changes with each subsequent call of the method.

This experiment was conducted on the Barbora GPU node and also on the laptop. The results on the two machines for the single layer matrix V_h are shown in Tables 11 and 12, results on other matrices were similar. On the Barbora GPU node we utilized all 4 GPUs and 24

Table 10: Comparison of computation times using the four GPU algorithm versions (in seconds)

(a) Single layer matrix V_h

E_t	64	128	256	512
E_s	1536	3072	6144	12288
Version 1	0.081	0.579	5.347	70.085
Version 2	0.053	0.474	3.709	55.032
Version 3	0.226	1.345	8.891	135.378
Version 4	0.174	0.829	4.355	51.913

(b) Double layer matrix K_h

E_t	64	128	256	512
E_s	1536	3072	6144	12288
Version 1	0.068	0.682	7.865	113.068
Version 2	0.103	0.697	5.868	68.270
Version 3	0.249	1.554	10.450	159.384
Version 4	0.205	1.067	5.984	71.589

(c) Adjoint double layer matrix $\mathsf{K}_h^{\top_s}$

E_t	64	128	256	512
E_s	1536	3072	6144	12288
Version 1	0.078	0.848	10.763	154.465
Version 2	0.094	0.605	4.681	65.664
Version 3	0.303	1.892	13.229	198.940
Version 4	0.211	0.989	5.227	60.302

(d) Hypersingular matrix D_h

E_t	64	128	256	512
E_s	1536	3072	6144	12288
Version 1	0.249	2.230	24.816	352.307
Version 2	0.176	1.220	9.697	104.010
Version 3	0.673	4.080	25.983	366.300
Version 4	0.233	1.292	11.958	132.642

processor cores, using a mesh with 512 temporal and 12288 spatial elements. On the laptop we utilized all 8 processor cores and used a mesh with 128 temporal and 3072 spatial elements. No warm-up runs were performed this time.

On the Barbora GPU node the performance benefit of using the CPU-GPU load balancing was very small, achieving at most 2% increase in performance, with the average speedup over the 10 iterations being 1.013. From the number of elements assigned to the CPU in each iteration we can notice, that the load-balancer has a tendency to oscillate. This behavior was also observed for other mesh refinements and matrices and is probably caused by non-linear scaling of the CPU implementation.

The results from the laptop are much more interesting. The CPU-GPU load-balancer was able to speed up the matrix-vector multiplication by 22 %, with the average over the 10 iterations being 19 %.

The reason why the computation time decreases by a larger amount on the laptop is the relative performance of the CPU and GPU. On the Barbora GPU node the performance of the GPUs in the machine absolutely dominates, while on the laptop the GPU performance is not that high when compared to the CPU.

6.10 Performance comparison of accelerated and original code

Knowing the best configurations and properties of the CPU and GPU on-the-fly implementations, we can finally compare them with each other and with the original in-memory approach. We measure the time of a single matrix-vector multiplication, as well as the total time of solving the whole Dirichlet or Neumann problem. To save resources, we reduced the number of repeated executions of the methods.

Table 11: CPU-GPU load balancing on the Barbora GPU node

Iteration	CPU elements count	CPU elements ratio [%]	Time [s] CPU	Time [s] GPU	Time [s] Total	Speedup
1	24	0.2	11.79	55.06	55.24	1.00
2	336	2.7	49.89	53.97	54.09	1.02
3	367	3.0	55.28	53.67	55.38	0.99
4	354	2.9	52.96	53.92	54.03	1.02
5	361	2.9	54.06	53.70	54.16	1.02
6	358	2.9	52.77	53.96	54.06	1.02
7	367	3.0	55.18	53.91	55.28	0.99
8	357	2.9	52.57	53.93	54.03	1.02
9	367	3.0	54.91	53.74	55.00	1.00
10	358	2.9	52.63	53.95	54.05	1.02

Table 12: CPU-GPU load balancing on the laptop

Iteration	CPU elements count	CPU elements ratio [%]	Time [s] CPU	Time [s] GPU	Time [s] Total	Speedup
1	8	0.3	2.24	91.19	91.19	1.00
2	590	19.2	78.14	78.14	78.15	1.16
3	568	18.5	75.02	75.03	75.03	1.21
4	566	18.4	74.97	74.97	74.98	1.21
5	563	18.3	74.61	74.61	74.61	1.22
6	563	18.3	74.14	74.61	74.61	1.22
7	565	18.4	74.48	74.61	74.61	1.22
8	565	18.4	75.25	75.25	75.26	1.21
9	561	18.3	74.47	74.61	74.61	1.22
10	561	18.3	73.74	74.61	74.61	1.22

Matrix-vector multiplication

The measured times of the execution of the `apply` method (in seconds) for all four main boundary element matrices and two mesh refinements are shown in Table 13. The measurements of the original and the CPU on-the-fly implementations were carried out on the Barbora CPU node, the GPU on-the-fly implementation was measured on the Barbora GPU node. The experiment was also conducted on the laptop, the results are shown in Table 14. On all three machines we utilized all available processor cores and GPUs.

Using the original in-memory approach, the assembly of the matrices usually takes the most of the time, the multiplication of the matrix with a vector then takes only a fraction of the assembly time. However, as we can see in the table, on the Barbora CPU node for the 256×3144 mesh refinement the matrix-vector multiplication took significantly more time than just a fraction (except for D_h). A partial explanation of the decreased performance is the NUMA effect. The fact that the matrices V_h, K_h take 72 and 36 GB of memory, respectively, could also be a reason

Table 13: Computation times of all three implementations of the `apply` method on the Barbora nodes, in seconds

Mesh refinement	$E_t = 256$, $E_s = 6144$				$E_t = 128$, $E_s = 3072$			
Matrix	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h
In-memory, assembly	134.72	146.98	146.98	581.27	17.10	19.43	19.43	75.75
In-memory, multipl.	150.95	71.38	70.55	9.44	2.51	1.24	0.62	0.21
In-memory, total	285.67	218.36	217.53	590.70	19.62	20.67	20.06	75.95
CPU on-the-fly	92.59	127.65	147.81	294.58	9.78	12.62	14.60	27.70
GPU on-the-fly	3.86	6.02	4.79	9.95	0.50	0.73	0.64	1.25
CPU on-the-fly speedup	3.09	1.71	1.47	2.01	2.01	1.64	1.37	2.74
GPU on-the-fly speedup	74.07	36.29	45.40	59.39	39.35	28.43	31.29	60.83

Table 14: Computation times of all three implementations of the `apply` method on the laptop, in seconds

Mesh refinement	$E_t = 64$, $E_s = 1536$				$E_t = 32$, $E_s = 768$			
Matrix	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h	V_h	K_h	$\mathsf{K}_h^{\top_s}$	D_h
In-memory, assembly	55.00	56.88	56.88	75.30	6.96	7.42	7.42	9.64
In-memory, multipl.	1.64	0.79	0.77	0.34	0.09	0.05	0.04	0.02
In-memory, total	56.64	57.67	57.65	75.63	7.05	7.46	7.46	9.66
CPU on-the-fly	49.01	52.57	52.72	70.77	6.27	6.63	6.64	8.85
GPU on-the-fly	9.43	11.05	11.12	14.38	1.20	1.42	1.41	1.84
CPU on-the-fly speedup	1.16	1.10	1.09	1.07	1.13	1.13	1.12	1.09
GPU on-the-fly speedup	6.01	5.22	5.19	5.26	5.90	5.25	5.29	5.26

for this slowdown.

Looking at the computation times of the CPU on-the-fly implementation we observe, that it is always faster than the in-memory approach. On the laptop the difference is at most 16 %, on the Barbora CPU node the speedup is more significant, with the CPU on-the-fly implementation being around 1.5–3 times faster. The speedup of the CPU and GPU on-the-fly versions is measured with respect to the total in-memory time.

On the laptop the GPU on-the-fly implementation is 5–6 times faster than the original code. On the Barbora nodes the GPU on-the-fly implementation is 30–70 times faster than the original approach depending on the matrix.

Solution of the Dirichlet and Neumann problems

We now compare the computational times needed to solve the Dirichlet or Neumann problem for the heat equation. We compare only the results of the in-memory approach and the GPU on-the-fly implementation, since the CPU on-the-fly version performed very poorly. The timing results for several mesh refinements are shown in Table 15 for the Barbora nodes and in Table 16 for the laptop.

Table 15: Execution times (in seconds) of solving the Dirichlet and Neumann problems on the Barbora nodes

			Dirichlet problem				Neumann problem			
E_t	E_s	Implem.	prep.	solve	total	spdp.	prep.	solve	total	spdp.
16	384	mem	0.11	0.04	0.14	–	0.24	0.02	0.26	–
		GPU	0.57	0.11	0.67	0.21	0.56	0.24	0.81	0.32
32	768	mem	0.67	0.41	1.08	–	1.57	0.19	1.76	–
		GPU	0.59	0.46	1.05	1.03	0.58	1.42	2.00	0.88
64	1536	mem	4.86	7.08	11.94	–	11.42	1.30	12.72	–
		GPU	0.66	2.78	3.44	3.47	0.70	8.62	9.32	1.36
128	3072	mem	37.77	121.74	159.51	–	87.14	15.92	103.06	–
		GPU	1.26	26.64	27.91	5.72	1.32	78.14	79.46	1.30
256	6144	mem	369.26	8320.26	8689.52	–	756.64	748.54	1505.17	–
		GPU	6.55	227.64	234.18	37.11	5.34	756.30	761.64	1.98
512	12288	mem	–	–	–	–	–	–	–	–
		GPU	69.40	3458.35	3527.76	–	66.51	9893.99	9960.50	–

Table 16: Execution times (in seconds) of solving the Dirichlet and Neumann problems on the laptop

			Dirichlet problem				Neumann problem			
E_t	E_s	Implem.	prep.	solve	total	spdp.	prep	solve	total	spdp.
16	384	mem	2.05	0.14	2.20	–	2.41	0.04	2.45	–
		GPU	1.42	4.30	5.72	0.38	1.41	6.51	7.92	0.31
32	768	mem	15.89	3.40	19.29	–	18.63	0.79	19.42	–
		GPU	2.93	43.13	46.06	0.42	2.95	74.27	77.21	0.25
64	1536	mem	126.94	70.70	197.64	–	146.54	17.15	163.69	–
		GPU	14.99	416.07	431.06	0.46	14.98	708.80	723.77	0.23
128	3072	mem	–	–	–	–	–	–	–	–
		GPU	112.70	3872.86	3985.55	–	112.62	7731.11	7843.73	–

We measured the preprocessing time (matrix assembly, mesh copy to GPU, right-hand-side vector assembly) and the time of solving the system using the FGMRES algorithm, which utilizes the `apply` method. Further we show the total time and the speedup expressing the relative performance of the GPU implementation compared to the in-memory approach. The relative accuracy of the FGMRES method was set to 10^{-8}. All available GPUs and processor cores were utilized.

From the tables we can conclude, that on the Barbora nodes the GPU implementation is faster for finer discretizations of the mesh. For the Neumann problem the speedup was no more than 2. For the Dirichlet problem we got a maximum speedup of 37, but this is not much relevant because of the mentioned issue with the single layer matrix multiplication. One would expect a speedup around 8.

On the laptop the GPU implementation needs more time to solve the problem than the in-memory approach. The reason to this is the lower performance of the GPU relative to the CPU. The time-consuming matrix assembly creates even larger difference in computation time between the preprocessing and the solution itself for both the Dirichlet and the Neumann problem.

The main advantage of the GPU implementation is, that it enables us to solve larger problems that would not fit into memory in the case of the original approach. For example, on the Barbora nodes we were not able to solve the Dirichlet problem with the 512×12288 mesh using in-memory approach, because the single layer matrix would occupy 576 GB of memory (maximum memory capacity of the node is 192 GB). However, the GPU on-the-fly approach was able to solve the problem in just under an hour.

6.11 Convergence

In the final experiment we calculate the relative error of the solution for several refinements of the mesh and estimate the order of convergence. For comparison we solve the same problem as in [30]. We solve the Dirichlet and Neumann problems with a known solution, both having the Dirichlet data

$$u(\boldsymbol{x}, t) = G_\alpha(\boldsymbol{x} - \boldsymbol{y}^*, t) \quad \text{for } (\boldsymbol{x}, t) \in \Sigma$$

and the Neumann data

$$w(\boldsymbol{x}, t) = \alpha \frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{x}, t) = \alpha \frac{\partial G_\alpha}{\partial \boldsymbol{n}_{\boldsymbol{x}}}(\boldsymbol{x} - \boldsymbol{y}^*, t) \quad \text{for } (\boldsymbol{x}, t) \in \Sigma$$

with $\boldsymbol{y}^* = (0, 0, 1.5)$. We solve the problems in the space-time domain $Q = (-1, 1)^3 \times (0, 1)$ and choose the heat capacity constant $\alpha = 0.5$. The emerging system of linear equations is solved using the FGMRES method with a relative accuracy of 10^{-8}.

The relative L^2 error is expressed as

$$L^2(\Sigma_h)(u_h) = \frac{\|u - u_h\|_{L^2(\Sigma_h)}}{\|u\|_{L^2(\Sigma_h)}}$$

with

$$\|u\|^2_{L^2(\Sigma_h)} = \langle u, u \rangle_{\Sigma_h} = \int_0^T \int_{\Gamma_h} |u(\boldsymbol{x}, t)|^2 \, \mathrm{d}\boldsymbol{s}_{\boldsymbol{x}} \, \mathrm{d}t,$$

and is calculated using standard quadrature rules. The estimated order of convergence is calculated as

$$\mathrm{eoc}(u_h) = \log_2 \left(\frac{L^2(\Sigma_h)(u_{2h})}{L^2(\Sigma_h)(u_h)} \right).$$

The results are shown in Table 17 for fixed $h_{\boldsymbol{x}}^2/h_t$ and in Table 18 for keeping constant $h_{\boldsymbol{x}}/h_t$.

We can see, that when keeping the ratio $h_{\boldsymbol{x}}^2/h_t$ fixed, we achieve higher order of convergence than with fixed $h_{\boldsymbol{x}}/h_t$. The obtained estimated orders of convergence agree with those observed in [14, 30].

Table 17: Convergence results, $h_{\boldsymbol{x}}^2 \approx h_t$

		Dirichlet problem		Neumann problem	
E_t	E_s	$L^2(\Sigma_h)$	eoc	$L^2(\Sigma_h)$	eoc
8	192	6.08e-01	–	3.14e-01	–
32	768	2.65e-01	1.198	8.64e-02	1.862
128	3072	1.13e-01	1.227	2.10e-02	2.038
512	12288	5.25e-02	1.109	5.01e-03	2.070

Table 18: Convergence results, $h_{\boldsymbol{x}} \approx h_t$

		Dirichlet problem		Neumann problem	
E_t	E_s	$L^2(\Sigma_h)$	eoc	$L^2(\Sigma_h)$	eoc
8	192	6.08e-01	–	3.14e-01	–
16	768	4.28e-01	0.506	1.51e-01	1.058
32	3072	1.80e-01	1.248	6.88e-02	1.131
64	12288	9.94e-02	0.858	3.45e-02	0.994

7 Conclusion

In this thesis we briefly introduced the space-time boundary element method for the heat equation. Then we made an overview of the current implementation of the BESTHEA library. We accelerated the code and explained the principles of its functionality and finally conducted several numerical experiments focusing mainly on the different implementation approaches.

We implemented a CPU code for the on-the-fly matrix-vector multiplication, which for a single use of the matrix achieves better performance than the original in-memory approach. The code was then accelerated for GPUs using CUDA, and is able to utilize all GPUs in a multi-GPU environment. The accelerated version achieved a speedup in the order of tens for a single matrix-vector multiplication compared to the original in-memory approach, the speedup of solving the Dirichlet and Neumann problems was approximately 8 and 2, respectively. Most importantly, the on-the-fly GPU-accelerated implementation enables us to solve larger problems, which we were previously unable to solve due to large memory requirements of the original in-memory implementation. Moreover, we implemented CPU-GPU load balancing, which further reduces the time needed to solve the problem.

We explored several optimizations of the developed algorithms to increase their performance. The list of possible optimization is, however, far from exhausted. In future work we plan to implement and test several other optimizations.

The new LUMI supercomputer in Finland [12], which is expected to be the most powerful supercomputer at the time of its launch, will draw most of its performance from AMD GPUs. We expect this decision to cause a shift in the HPC industry into using more portable code across the AMD and Nvidia GPUs. The proposed way of writing such portable applications is to use HIP (Heterogeneous-Computing Interface for Portability) developed by AMD, which is very similar to CUDA, but can run on both AMD and Nvidia GPUs [1, 13]. Converting the accelerated part of the BESTHEA library from CUDA to HIP will be a part of future work.

References

[1] Advanced Micro Devices, Inc. HIP Programming Guide. Available at https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_Programming_Guide_v4.1.pdf, 2021. Cited 2021-04-20.

[2] Betcke, T., and Scroggs, M. W. Bempp-cl: A fast Python based just-in-time compiling boundary element library. *Journal of Open Source Software 6*, 59 (2021), 2879.

[3] CMake community. CMake. Available at https://cmake.org/.

[4] Dohr, S. *Space-time boundary element methods for the heat equation. Diploma thesis.* Technischen Universität Graz, 2016.

[5] Harbrecht, H., and Zaspel, P. A scalable H-matrix approach for the solution of boundary integral equations on multi-GPU clusters. *arXiv preprint arXiv:1806.11558* (2018).

[6] ISO/IEC. *ISO International Standard ISO/IEC 14882:2017(E) – Programming Language C++.* Geneva, Switzerland: International Organization for Standardization (ISO), 2017.

[7] IT4Innovations National Supercomputing Center. IT4Innovations Documentation. Available at https://support.it4i.cz/. Cited 2021-03-28.

[8] Khronos OpenCL Working Group. The OpenCL Specification. Available at https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, 2020. Cited 2021-04-20.

[9] Kirk, D. B., and Wen-Mei, W. H. *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann, 2016.

[10] Lions, J. L., and Magenes, E. *Non-homogeneous boundary value problems and applications: Vol. 2*, vol. 182. Springer-Verlag, Berlin, 1972.

[11] Lions, J. L., and Magenes, E. *Non-homogeneous boundary value problems and applications: Vol. 1*, vol. 181. Springer Science & Business Media, 2012.

[12] Manninen, P., Robertsén, F., and Markomanolis, G. May we introduce: LUMI. Available at https://www.lumi-supercomputer.eu/may-we-introduce-lumi/. Cited 2021-04-19.

[13] Markomanolis, G., and Robertsén, F. Preparing codes for LUMI: converting CUDA applications to HIP. Available at https://www.lumi-supercomputer.eu/preparing-codes-for-lumi-converting-cuda-applications-to-hip/. Cited 2021-04-26.

[14] Messner, M. *A fast multipole Galerkin boundary element method for the transient heat equation.* Graz University of Technology, 2014.

[15] Meuer, H., Strohmaier, E., Dongarra, J., and Simon, H. Top500 supercomputer sites. Available at https://www.top500.org/, 2001. Cited 2021-04-19.

[16] NVIDIA Corporation. CUDA C++ Programming guide. Available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Cited 2021-04-19.

[17] Of, G., Merta, M., Zapletal, J., and Watschinger, R. BESTHEA library. Available at https://sites.google.com/view/besthea/. Cited 2021-04-04.

[18] Of, G., and Watschinger, R. A partial integration formula for the bilinear form of the hypersingular boundary integral operator of the heat equation in 3d. *In preparation* (2021).

[19] OpenACC-Standard.org. The OpenACC Application Programming Interface. Available at https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf, 2019. Cited 2021-04-20.

[20] OpenMP Architecture Review Board. OpenMP Application Programming Interface. Available at https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, 2020. Cited 2021-04-20.

[21] Rjasanow, S., and Steinbach, O. *The fast solution of boundary integral equations*. Springer Science & Business Media, 2007.

[22] Saad, Y. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing 14*, 2 (1993), 461–469.

[23] Sauter, S. A., and Schwab, C. Boundary element methods. In *Boundary Element Methods*. Springer, 2010, pp. 183–287.

[24] Takahashi, T., and Hamada, T. GPU-accelerated boundary element method for Helmholtz equation in three dimensions. *International Journal for Numerical Methods in Engineering 80*, 10 (2009), 1295–1321.

[25] Vater, K., Betcke, T., and Dilba, B. Simple and efficient GPU parallelization of existing H-Matrix accelerated BEM code. *arXiv preprint arXiv:1711.01897* (2017).

[26] Wang, Y., Wang, Q., Deng, X., Xia, Z., Yan, J., and Xu, H. Graphics processing unit (GPU) accelerated fast multipole BEM with level-skip M2L for 3D elasticity problems. *Advances in Engineering Software 82* (2015), 105–118.

[27] Watschinger, R., Merta, M., Zapletal, J., and Of, G. A parallel fast multipole method for space-time boundary element method for the heat equation. *In preparation* (2021).

[28] Wikipedia. Nvidia Tesla. Available at https://en.wikipedia.org/wiki/Nvidia_Tesla. Cited 2021-04-27.

[29] Zapletal, J., Merta, M., and Malý, L. Boundary element quadrature schemes for multi- and many-core architectures. *Computers & Mathematics with Applications 74*, 1 (2017), 157–173.

[30] Zapletal, J., Watschinger, R., Of, G., and Merta, M. Semi-analytic integration for a parallel space-time boundary element method modeling the heat equation. *arXiv preprint arXiv:2102.09811* (2021).

A Contents of the attachment

All the source codes of the BESTHEA library, the experiment programs, scripts and results are attached to this thesis in the electronical form. The important nodes in the directory structure inside the attached `.zip` file are following:

- `build` – currently empty directory in which the source code should be built
- `examples` – directory containing source codes of several examples of usage of the library
- `experiments`
  - `_results` – directory with all experiment results
  - `_scripts` – directory with the scripts used to run the experiments
  - other folders contain source codes of the experiments
- `include` – directory with all `*.h` files of the BESTHEA library
- `src` – directory with all `*.cpp` and `*.cu` source codes of the library
- `CMakeLists.txt` – root CMakeLists file used in the compilation
- `compilation_instructions.txt` – instructions for compilation
- `directory_structure.txt` – a text file containing this list

B Additional code listings

```
template< int quadr_order >
__global__ void g_apply_gpu_treg_sreg_ver2
  ( [[maybe_unused]] const besthea::bem::spacetime_heat_sl_kernel_antiderivative * _hka,
    [[maybe_unused]] const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >
     * _tst_space,
    [[maybe_unused]] const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >
     * _trl_space,
    const sc * x_perm, lo ld_x_perm, sc * y, lo ld_y, sc alpha, lo i_tst_begin,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_metadata mesh_metadata,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_data mesh_data,
    const ns_gpu_helpers::heat_kernel_parameters kp) {

  constexpr int tpbx = tpb_V[2][quadr_order].x;

  __shared__ ns_gpu_helpers::quadrature_nodes_raw<quadr_order> shmem_quadr_nodes_tst;
  __shared__ sc shmem_matrix_vals[tpbx];

  const lo &n_blocks = mesh_metadata.n_temporal_elements;
  const lo &n_elems = mesh_metadata.n_elems;
  const unsigned int &tid = threadIdx.x;
  const lo i_tst = i_tst_begin + blockIdx.x;
  const lo &row = i_tst;

  d_triangles_to_geometry_000_tst_shmem(i_tst, mesh_data, shmem_quadr_nodes_tst);
  __syncthreads();

  ns_gpu_helpers::quadrature_nodes_raw<quadr_order> quadr_nodes_trl;


  sc val_prev;
  sc val_curr;
  sc val_next;

  for(lo i = threadIdx.x; i < n_elems; i += blockDim.x) {
    d_triangles_to_geometry_000_trl(i, mesh_data, quadr_nodes_trl);

    val_curr = 0;
    val_next = 0;

    lo curr_active_threads =
      (i >= (n_elems / blockDim.x) * blockDim.x) ? (n_elems % blockDim.x) : blockDim.x;

    for(lo delta = 0; delta < n_blocks; delta++) {
      // each thread calculates value corresponding to its i (i_trl)
      {
        lo &i_trl = i;
        val_prev = val_curr;
        val_curr = val_next;
        d_get_local_contributions_treg_sreg_sl_p0_p0(&val_next, delta+1, i_tst, i_trl,
          shmem_quadr_nodes_tst, quadr_nodes_trl, mesh_metadata, mesh_data, kp);
        shmem_matrix_vals[tid] = ((i_tst == i_trl) ? (0) : (-val_prev + 2*val_curr - val_next));
        __syncthreads();
      }

      // now the thread logic is changed, each thread takes one (or more) blocks
      // in this delta and loops through all of the current trial elements (columns)
      {
        lo max_block = n_blocks - delta;
        for(lo block = threadIdx.x; block < max_block; block += curr_active_threads) {
          lo block_row = delta + block;
          lo block_col = block;
          sc y_val = 0;
          for(lo j = 0; j < curr_active_threads; j++) {
            lo i_trl = (i / blockDim.x) * blockDim.x + j;
            lo &col = i_trl;
            sc x_val = x_perm[block_col + ld_x_perm * col];
            y_val += shmem_matrix_vals[j] * x_val;
          }
          y_val *= alpha;
          y[block_row * ld_y + row] += y_val;
        }
        __syncthreads();
      }

    }

  }

}
```

Listing 12: GPU on-the-fly algorithm version 2

```
template< int quadr_order >
__global__ void g_apply_gpu_treg_sreg_ver3
  ( [[maybe_unused]] const besthea::bem::spacetime_heat_sl_kernel_antiderivative * _hka,
    [[maybe_unused]] const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >
     * _tst_space,
    [[maybe_unused]] const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >
     * _trl_space,
    const sc * x_perm, lo ld_x_perm, sc * y, lo ld_y, sc alpha, lo i_tst_begin,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_metadata mesh_metadata,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_data mesh_data,
    const ns_gpu_helpers::heat_kernel_parameters kp) {

  constexpr int tpbx = tpb_V[3][quadr_order].x;
  constexpr int tpby = tpb_V[3][quadr_order].y;

  __shared__ ns_gpu_helpers::quadrature_nodes_raw<quadr_order> shmem_quadr_nodes_tst[tpbx];
  __shared__ ns_gpu_helpers::quadrature_nodes_raw<quadr_order> shmem_quadr_nodes_trl[tpby];
  __shared__ sc shmem_y_vals[tpbx * tpby];

  const int tid = threadIdx.y * blockDim.x + threadIdx.x;
  const lo &n_blocks = mesh_metadata.n_temporal_elements;
  const lo &n_elems = mesh_metadata.n_elems;
  const lo i_tst = i_tst_begin + blockIdx.x * blockDim.x + threadIdx.x;

  shmem_y_vals[tid] = 0;
  if(tid < blockDim.x)
    d_triangles_to_geometry_000_tst(i_tst_begin + blockIdx.x * blockDim.x + tid, mesh_data,
     shmem_quadr_nodes_tst[tid]);
  __syncthreads();

  sc matrix_val;
  sc val_prev;
  sc val_curr;
  sc val_next;

  for(lo i_trl = threadIdx.y; i_trl < n_elems; i_trl += blockDim.y) {
    if(tid < blockDim.y)
      d_triangles_to_geometry_000_trl((i_trl / blockDim.y) * blockDim.y + tid, mesh_data,
     shmem_quadr_nodes_trl[tid]);
    __syncthreads();

    const lo &row = i_tst;
    const lo &col = i_trl;

    val_curr = 0;
    val_next = 0;

    for(lo delta = 0; delta < n_blocks; delta++) {
      val_prev = val_curr;
      val_curr = val_next;
      d_get_local_contributions_treg_sreg_sl_p0_p0(&val_next, delta+1, i_tst, i_trl,
        shmem_quadr_nodes_tst[threadIdx.x], shmem_quadr_nodes_trl[threadIdx.y], mesh_metadata,
     mesh_data, kp);
      matrix_val = ((i_tst == i_trl) ? (0) : (-val_prev + 2*val_curr - val_next));

      lo max_block = n_blocks - delta;
      for(lo block = 0; block < max_block; block++) {
        lo block_row = delta + block;
        lo block_col = block;
        shmem_y_vals[tid] = matrix_val * x_perm[block_col + ld_x_perm * col];
        __syncthreads();
        d_reduce_sum_2d_y<tpbx,tpby>(shmem_y_vals);
        if(tid < blockDim.x) y[block_row * ld_y + row] += alpha * shmem_y_vals[tid];
        __syncthreads();
      }

    }

    shmem_y_vals[tid] = 0;
    __syncthreads();

  }

}
```

Listing 13: GPU on-the-fly algorithm version 3

```
template< int quadr_order >
__global__ void g_apply_gpu_treg_sreg_ver4
  ( [[maybe_unused]] const besthea::bem::spacetime_heat_sl_kernel_antiderivative * _hka,
    [[maybe_unused]] const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >
     * _tst_space,
    [[maybe_unused]] const besthea::bem::uniform_spacetime_be_space< besthea::bem::basis_tri_p0 >
     * _trl_space,
    const sc * x_perm, lo ld_x_perm, sc * y, lo ld_y, sc alpha, lo i_tst_begin,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_metadata mesh_metadata,
    const besthea::mesh::uniform_spacetime_tensor_mesh_gpu::mesh_raw_data mesh_data,
    const ns_gpu_helpers::heat_kernel_parameters kp) {

  constexpr int tpbx = tpb_V[4][quadr_order].x;
  constexpr int tpby = tpb_V[4][quadr_order].y;

  __shared__ ns_gpu_helpers::quadrature_nodes_raw<quadr_order> shmem_quadr_nodes_tst[tpbx];
  __shared__ ns_gpu_helpers::quadrature_nodes_raw<quadr_order> shmem_quadr_nodes_trl[tpby];
  __shared__ sc shmem_matrix_vals[tpbx * tpby];

  const int tid = threadIdx.y * blockDim.x + threadIdx.x;
  const lo &n_blocks = mesh_metadata.n_temporal_elements;
  const lo &n_elems = mesh_metadata.n_elems;
  const lo i_tst = i_tst_begin + blockIdx.x * blockDim.x + threadIdx.x;
  const lo &row = i_tst;

  if(tid < blockDim.x)
    d_triangles_to_geometry_000_tst(i_tst_begin + blockIdx.x * blockDim.x + tid, mesh_data,
     shmem_quadr_nodes_tst[tid]);
  __syncthreads();

  sc val_prev;
  sc val_curr;
  sc val_next;

  for(lo i = threadIdx.y; i < n_elems; i += blockDim.y) {
    if(tid < blockDim.y)
      d_triangles_to_geometry_000_trl((i / blockDim.y) * blockDim.y + tid, mesh_data,
     shmem_quadr_nodes_trl[tid]);
    __syncthreads();

    val_curr = 0;
    val_next = 0;

    lo curr_active_threads = (i >= (n_elems / blockDim.y) * blockDim.y) ? (n_elems % blockDim.y) :
      blockDim.y;

    for(lo delta = 0; delta < n_blocks; delta++) {
      {
        lo &i_trl = i;
        val_prev = val_curr;
        val_curr = val_next;
        d_get_local_contributions_treg_sreg_sl_p0_p0(&val_next, delta+1, i_tst, i_trl,
          shmem_quadr_nodes_tst[threadIdx.x], shmem_quadr_nodes_trl[threadIdx.y], mesh_metadata,
     mesh_data, kp);
        shmem_matrix_vals[tid] = ((i_tst == i_trl) ? (0) : (-val_prev + 2*val_curr - val_next));
        __syncthreads();
      }

      {
        lo max_block = n_blocks - delta;
        for(lo block = threadIdx.y; block < max_block; block += curr_active_threads) {
          lo block_row = delta + block;
          lo block_col = block;
          sc y_val = 0.0;
          for(int j = 0; j < curr_active_threads; j++) {
            lo i_trl = (i / blockDim.y) * blockDim.y + j;
            lo &col = i_trl;
            sc x_val = x_perm[block_col + ld_x_perm * col];
            y_val += shmem_matrix_vals[j * blockDim.x + threadIdx.x] * x_val;
          }
          y_val *= alpha;
          y[block_row * ld_y + row] += y_val;
        }
        __syncthreads();
      }

    }

  }

}
```

Listing 14: GPU on-the-fly algorithm version 4